Embedded Instruction Kit

A Major Qualifying Project Report
submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfilment of the requirements for the
Degree of Bachelor of Science by:

Steven Murdy
Alexander Dymek

Date: June 30, 2016

Report Submitted to:

Professor Stephen Bitar
Worcester Polytechnic Institute

This report represents the work of two WPI undergraduate students submitted to the faculty as evidence of completion of a degree requirement. WPI routinely publishes these reports on its website without editorial or peer review.

Abstract

The purpose of this project was to create an electronics kit for teaching entry level programming of embedded systems. The kit includes a development board, external hardware modules, software, and teaching materials. The hardware modules consist of a LCD display, Keypad and various user I/O. These modules were then incorporated into lessons for creating several working projects including a Calculator, Clock, Morse Code Interpreter, a Maze Game and Simon Says.

Table of Contents

- Abstract
- Table of Contents
- Acronym Reference Table
- 1 Introduction
- 2 Background
  - 2.1 CPU and Microprocessors
  - 2.2 Microcontrollers
  - 2.3 Existing Kits
- 3 Methodology
  - 3.1 Initial Planning
  - 3.2 Designing the Lessons
  - 3.3 Development Board
  - 3.4 The Modules
  - 3.5 Writing and Testing the Lessons
- 4 Results
- 5 Conclusions
- Works Cited
- Appendix A - Introductory Lesson
- Appendix B Calculator Lesson
- Appendix C Clock Lesson
- Appendix D Morse Code Interpreter Lesson
- Appendix E Maze Game Lesson
- Appendix F Simon Says Lesson
- Appendix G Keypad Module Circuit Diagram
- Appendix H User I/O Module Circuit Diagram

Acronym Reference Table

ACRONYM	MEANING
CS	Computer Science
ECE	Electrical and Computer Engineering
RISC	Reduced Instruction Set Computing
ARM	Acorn RISC Machine
WPI	Worcester Polytechnic Institute
I/O	Input/output
PC	Personal Computer
IC	Integrated Circuit
PROM	Programmable Read Only Memory
EPROM	Erasable PROM
EEPROM	Electrically EPROM
ADC	Analog to Digital Converter
DAC	Digital to Analog Converter
IDE	Integrated Development Environment
DIY	Do It Yourself

1 Introduction

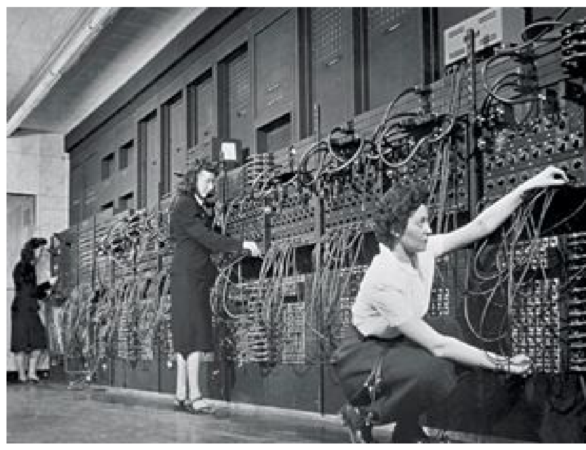
Embedded systems are integral part of modern life. They help get where we need to go by running many of the systems in cars and planes.They operate our watches, our phones, and our tablets. They control the robots that can manufacture a car, clean a house, or perform surgery. They even improve our everyday activities with e-readers for books, DVRs for TV, watches that record our exercise, and so many other devices. New uses for embedded systems are created every day, as manufacturers take complex tasks, and reduce them to the push of a button.

The purpose of this project was to create a kit meant to teach entry level embedded systems to buyers. There are three parts to the kit hardware, software, and teaching materials. The hardware component consists of a development board and a series of modules contain the various components needed during the course. The software component consists of the libraries needed to drive the development board and the modules. The teaching tools consist of the documentation of all parts of the project and a series of detailed lessons with designed to teach the basics of embedded systems. These lessons will have two phases the teaching phase where the students will learn new concepts, through a series of hands on assignments and the implementation phase where the students will use the learned concepts to complete an project. The projects are fully functioning applications - either serving some basic utility, or a working game - so that students will enjoy using their own work and take pride in developing on their own.

2 Background

2.1 CPU and Microprocessors

The core of computer circuitry is the central processing unit, or CPU, which carries out the logical, arithmetic, and control operations necessary to carry out the instructions of a computer program (*Zandbergen).* The CPU was first introduced with the implementation of "stored-program computers", computers that stored program instructions within electronic memory. The first electronic general purpose computer was the ENIAC (Electronic Numerical Integrator and Computer), built in February, 1946 (*Copeland*). The ENIAC and other computers like it had to be physically rewired to perform individual tasks, earning the moniker "fixed-program computers." Later computer models, such as the EDVAC (Electronic Discrete Variable Automatic Computer) were designed to perform instructions of various types without needing to modify the computer circuitry. Programs written for the computer were stored in computer memory, and modifying the memory would change what operation was performed. These early CPUs used relay switches and vacuum tubes as switching devices, and computers often needed thousand of these switching devices to run. As a result, early CPUs were fairly bulky in size.



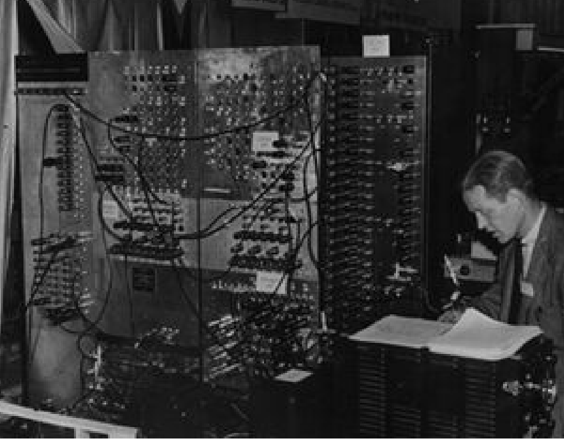


Figure 2.1 ENIAC,1946 Figure 2.2 EDVAC, 1951

With advancements in technology, CPUs could be made smaller, and could house more complex circuitry, allowing for multipurpose capabilities. Starting in the 1960s, transistors, semiconductor devices that could amplify or switch electronic signals, were used to replace the bulkier and fragile vacuum tubes. Circuits of interconnected transistors could be manufactured in a compact space, and then printed on a single semiconductor-based chip (to*mshardware.com*). These “integrated circuits” started as basic, non-specialized circuits, and CPUs would need a numerous quantity of integrated circuits to run. Additional transistors could be added to the integrated circuits, improving their capabilities even further.

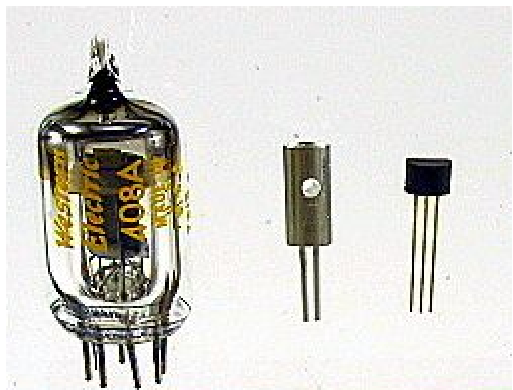


Figure 2.3 Size Comparison between Vacuum Tube(left) and Transistor(right)

Intel introduced their first microprocessors, the 4-bit Intel 4004 and 8-bit Intel 8008, in 1971 and 1972 respectively (*meetingtomorrow.com*).. In comparison to other CPUs which required multiple integrated circuits, microprocessors could run on a single integrated circuit, which greatly reduced the cost of processing power (*Singer*). Microprocessors were faster, more efficient, lighter, and a fraction of the size of previous CPUs. Their manageable size and efficiency has rendered all previous CPU models obsolete. Modern processors are even more efficient, operating on the nano level rather than the micro level, although, they are still referred to as microprocessors.

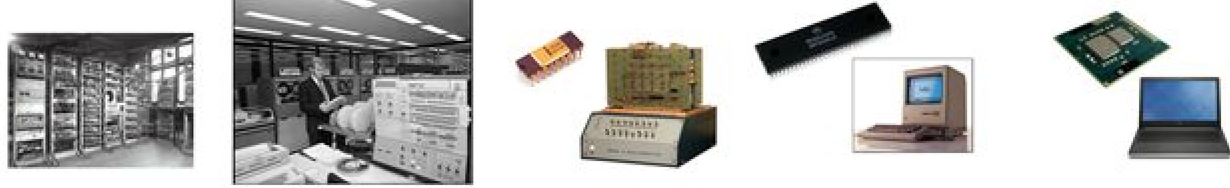


Figure 2.4 Comparison of Computer Designs, (from Left to Right, 1944 EDVAC, 1968 office computer, 1972 Microcomputer powered by Intel 8008, 1984 Apple Macintosh, 2013 Dell Insperion)

2.2 Microcontrollers

Microcontrollers are a self-contained system containing a processor, memory and peripherals. They are meant to be a cheaper, more cost effective solution for systems that need the abilities of a processor, but are not hardware intensive (*robotplatform.com*). For reference, a modern commercial processor could cost between $300 and $1000, while a microcontroller can be as little as a few cents.

The history of microcontrollers is tied to the history of the microprocessor. While the Intel 4004 was revolutionary in redesigning CPUs, the microprocessor required external memory, a motherboard and many other components, making any system that used them very expensive. The price of manufacturing made it cost-ineffective to use microprocessors when building appliances and other devices that weren't meant for major computation.

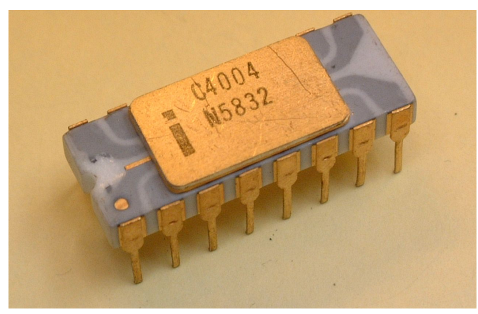


Figure 2.4 Intel 4004 Microprocessor

In 1974, Texas Instruments (TI) released the TMS 1000, which had read-only memory, read/write memory, processor and clock on one chip (*circuitstoday.com*). The cost required to produce this was greatly reduced in comparison to other microprocessors. Other companies, including Intel, followed suit by crafting their own microcontrollers, which were optimized for singular applications.

Figure 2.5 TMS1000 Microcontroller

The modern microcontrollers are a very diverse in both their specifications and their uses. They range from very fast, capable of running complex systems and devices such as smartphones, to one's meant to run lights. The diversity is a great as their applications in modern life: high power, low power, 8 bit, 64 bit, slow, fast, large memory, small. They now run most appliances, devices, toys, gadgets, and almost any other electrical system (*circuitstoday.com*).

2.3 Existing Kits

The idea of an educational tool designed to teach embedded computing is not a completely new idea. There are a number of kits that have been produced in the past that have been built for this exact purpose. The number of kits produced for this purpose is admittedly small, especially when viewed in comparison to educational kits devoted to other disciplines. Also, many of these kits are out of date, or have been discontinued, meaning that the market of embedded computing kits is virtually untouched.

There are a few theorized reasons why this market is so small. Embedded computing is a subject that is very in-depth and complex, more than electrical circuits or basic programming.

Starter kits are very expensive, and the software needs to be kept up-to-date to stay relevant. Most kits are not actually educational in nature, and assume the user is experienced with microcontrollers and has purchased the kit with the express purpose of building their own application, rather than learning how to program.

The first item for review is the *Thames & Kosmos Microcontroller Computer Systems Engineering Kit*. The kit includes a portable carrying case that has a main board and additional circuit components, a cable to connect the board to a computer, a CD containing the programming software, and an instruction booklet with directions for projects. The user needs to set up and wire the board by hand for each individual project, and the instructions for how to do so are contained within the manual.The product was discontinued in 2006 or 2007, and the software is not compatible with modern operating systems (www.parentschoice.org).

Figure 2.1: Thomas and Kosmos Microcontroller Computer Systems Engineering Kit

The *PICDEM Lab Development Kit* shares similarities with the *Thames & Kosmos* kit. The kit consists of a customizable board, an array of sensors and other circuit components, a CD containing software, a cable to connect to the board, and a virtual instruction manual with hands-on projects.The PICDEM kit has multiple PIC microcontrollers rather than a single microcontroller board. The kit is still available for purchase (www.microchip.com).

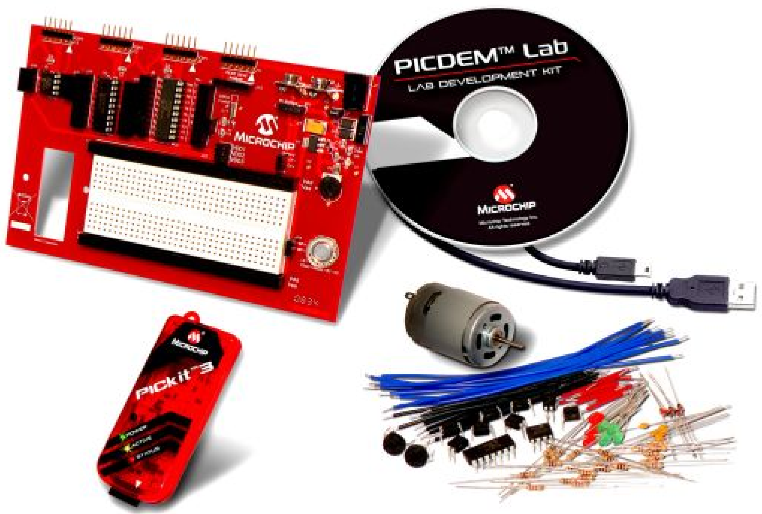


Figure 2.2 PICDEM Lab Development Kit

The *Nerdkits Microcontroller Kit* was developed by a team of MIT graduates to teach Electrical engineering and Software Development. The boards are composed of basic components that can be replaced easily. The kit includes an instruction guide that gives step-by-step instructions on how to perform the electrical wiring and how to write and compile programs. Sample projects are located on the website with videos on how to perform them. The kits are no longer being distributed (www.nerdkits.com).

Figure 2.3 Nerdkits Microcontroller Electronics Kit

The last kit researched was the *Getting Started with the BeagleBone Black Kit*, which features a the BeagleBone, a Linux-based microcontroller board with the programming capabilities of an Arduino-level microprocessor. The kit contains the BeagleBone board, an instruction manual with project applications, and an array of capacitors, resistors, LEDs and other circuit components(www.makershed.com).

Figure 2.4 Make: Getting Started with BeagleBone Kit

There were noticeable similarities between the different kits. Each kit was packaged as a disassembled unit, with a microcontroller, a solderless breadboard, and additional wires, resistors, LEDs, buttons, and sensors. The user had to connect the microcontroller to the breadboard, and wire the breadboard with the necessary electrical components needed to construct whatever application was presented.

Any of the other embedded kits that were found while researching were specialized kits used to design custom applications. These kits were meant to be used by experienced programmers, lacking instructional materials of any sort. Despite this, there was one kit that had a unique feature that caught our notice. The *Netduino Go Starter Kit* consisted of a main board with Netduino microprocessor, and 4 additional module boards that worked in conjunction with the main board. Each module had a few built in components, and could be easily connected to the board. This made the board more easy to use for a wide variety of applications (www.adafruit.com).

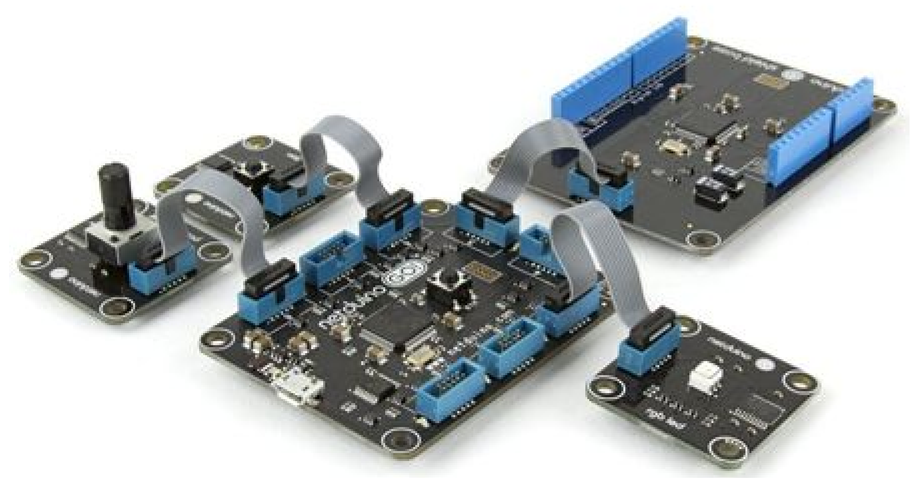


Figure 2.5 Netduino Go Starter Kit

Boards on market	**Circuitry**	**Board Type**	**Side Boards**	**Project**	**Manual**	**User Level**	**Software Location**	**Materials**
Thames& Kosmos	DIY	(n/a)	Single	100	Print	Begin	CD with Kit	Sensors
Nerdkits	DIY	Atmel	Single	26+	Online	Inter	Online Firmware	LCD, mosfets, sensors
PICDEM	DIY	PIC	Single	?	CD	Inter	CD with Kit	Mosfets, Motor, sensors
Beagle-Bone	DIY	ARM Cortex	Single	?	Print	Inter	Online Firmware	Sensors
Netduino Go	Prebuilt	Arduino	Five	N/A	N/A	Adv.	Online Firmware	5 modules
Ideal	***Prebuilt***	***ARM Cortex***	***4-5***	***12+***	***CD***	***Inter***	***Online/ CD***	***LCD, 4 Modules***

Figure 2.6 Comparison of Existing Educational Kits

3 Methodology

The methodology section will cover what we did during the course of the project. It will start with the initial meeting and planning of our project. It will then cover the selection and programming of our development board. Then it will discuss the design process of the extensions which became modules and the lessons we planned to teach. We will also discuss the construction and implementation of those modules and the choices we made. We will conclude with our creation and testing of the lessons.

3.1 Initial Planning

This project began when we separately came to Professor Bitar and proposed a kit dedicated to teaching students the basics of ECE. Alex proposed a kit that would teach student how to assemble basic hardware circuits, while Steven proposed a kit that would teach students the basics of embedded systems. Professor Bitar asked us meet together with him to see if we could combine our efforts and work together. After spending some time working together and brainstorming ideas, we decided to unify our project ideas. The final proposal was to create a kit that taught the basics of embedded programming and leave open the possibility of teaching students how to make and program components for an embedded circuit.

Now that we had decided what type of kit, we had to determine the scope of the project, we could not make a kit that would be useful to everyone. First we had to decide who our target demographic was. We debated what age group, and knowledge level the kit should be tailored to. The options for age group were divided by educational divisions: Middle School, High School, and College. We wanted the kit to teach the students skills that could be used outside the scope of the kit itself, and we wanted them to know and understand what they were doing, not just doing as they were told. As such we decided to aim the kit at High Schoolers, and early college students, as this group would understand the knowledge presented, while not be advanced enough that it would be too simple. The second part of this stage was to determine the level of

knowledge that the students should have coming into the project. Should we make a kit for someone who has no knowledge of programming, or an advanced knowledge? We decided to develop a kit for someone who has basic knowledge of programming C, but little to no experience with embedded systems. We did this because we wanted a kit to teach embedded programming, the creation of an application using both hardware and software. We did not want to have teach the students how to program from scratch. We use common programming terminology, which we did not want to teach. To summarize, we decided to build a kit for High School/College level students, who know how to program but wish to learn embedded systems, the basics of those systems.

Now that we knew who would be using the kit, and their background, we had to decide where the kit would be used. Was this a classroom tool, to be used by teachers to teach their student, or was it a DIY (Do it Yourself) Kit? We decided that this would be a DIY Kit, that would teach its user the topics. It could still be used by a teacher as a classroom tool, but we wanted it to be a commercial product for individual use, first and foremost.

The final step of of this stage was to determine what our kit should contain, and what we needed to deliver for our project. We split the kit into three types of deliverables: hardware, software, and lessons. For the hardware we would need a development board, that would serve as the center of our kit and an expansion(s) board that would add any hardware we wanted but was not included on the development board. For the software we would need to deliver a software library for the kit, complete with examples that would allow the student to use the hardware from the start. The lessons are the documents that would teach the students how to use the kit, and give them projects to do as they learn.

3.2 Designing the Lessons

We had the basic idea for the kit: who would use it, where, and what was should be in it. Now we had to determine how it would teach. As the the kit was meant to be DIY, we wanted to keep it interesting, as such we decided that we wanted to teach by application. We wanted to create a series of applications that would cover cover the material we wanted to teach, well giving to students a sense of accomplishment (and a new toy to play with). We then

brainstormed a series of applications that could be done by in an introductory course, and either had utility (a clock) or was a game (Simon Says). The results of this can be seen in Figure 3.1.

Application	Description	Topics
Hangman	2 Variants: 1. Multiplayer: 1 person enters the word, another solves the puzzle 2. Single Player: Randomly chooses a word and the player tries to guess the word	Buttons Graphical Display Memory Random Numbers
Tilt Maze	Move a ball through a maze by tilting sensor, possible random maze generation, randomly chosen maze from prebuilt mazes, level system (like pac-man)	Accelerometer Graphical Display ADC, Memory, Random Numbers, Buttons
Thermometer	Using the specs of a thermistor, determine the temperature of the room.	Thermistor, Display, ADC
Universal Clock	Use the microcontroller's clock to simulate a 12hr clock, complete with alarm, thermometer, timers, and stopwatch features.	Thermistor, Display, ADC, DAC, Clocks, Timers, Interrupts, Buttons, Speaker
Guitar Hero	Make a guitar hero game, play a song and light the LEDs corresponding to the current notes, the notes can descend the graphical display, choose songs	Graphical Display, DAC, Clocks, LEDs, Timers, Interrupts, Buttons, Memory, Speaker
Simon Says	Play the patterns on the LEDs and the speaker have the user repeat.	DAC, Clocks, Timers, Interrupts, LEDs, Speaker, Buttons
Morse Code	Have the user enter a morse code message that is translated onto the Display in real time.	Buttons, Clocks, Interrupts, Timers Graphical Display
Robotic Arm	Have a robotic arm that bats and incoming ball away	Optical Sensor, Cervos, DAC, ADC, Interrupts
Oscilloscope	Read in a wave form and display it in real time, Capture Mode, Storage	Graphical Display, ADC, Clocks, Timers, Interrupts, Buttons, Memory,
Voice Recorder	Read in sound through a microphone, store it, and play it back at the push of a button	Sampling, ADC, DAC, Microphone, Speaker, Memory, Buttons
Calculator	Make a basic calculator	Buttons, Graphical Display, Math

Figure 3.1 Brainstormed Projects

At this point we realised that we would have to teach the students the individual components one at a time, but we wanted to keep them learning by doing not by reading blocks of text. So we decided that to change the format from one large application, to a Lesson. A Lesson would be a series of Assignments, with some text explanations, that would culminate in a Project to build the application. This would allow the students a very hands on learning experience, while giving them the sense of building towards a goal: the application.

3.3 Development Board

We had a plan, we knew our target demographic, we knew what we wanted to teach, and how, now we had to begin building the kit. The first step was to choose a development board to use, as it is the central component of the kit and it is needed to develop any other components. We began by making a list of criteria for the board. The first criteria was that the board had to be very expandible, this would allow us to add any hardware our projects needed. The second criteria was usability, we did not want an over complicated microcontroller as that would make the kit harder to use. We also wanted the board to have minimal built in hardware, allowing it to change its functions based on the hardware it was connected to. The third criteria was versatility, we wanted the lessons that the students learned using the kit, and the board itself, to be usable in future projects. We wanted to get our students started on the path, and we wanted them to continue down it after.

The first step in choosing a board was to choose an architecture to use. We examined three architectures that we had used previously and had at least some experience with: ARM, Arduino, and MSP430. To decide which architecture was best suited to the purpose of our project we examined the prevalence of the architecture, its uses, the availability of software, and other criteria, see Figure 3.2 for more details.

Architecture	Advantages	Disadvantages
ARM	Large Range of Microcontrollers Large Range of Boards Commonly Used Usable in Wide Range of Applications	Difficult to Set Up Variety Limits Sample Usability
ARDUINO	Easy to Set Up Easy to Program Large Amount of Sample Code	Limited Applications Not C, but Arduino Language Not Commonly Used Limited Range of Microcontrollers Limited Range of Boards
MSP430™ Ultra-Low Power MCUs Texas Instruments	Usable in Range of Applications Some Sample Code	Some Limitations on Applications Not Commonly Used Limited Range of Microcontrollers Limited Range of Boards

Figure 3.2 Architecture Advantages and Disadvantages

MSP430 was the first architecture we ruled out, it had limitations in some applications, a small number of available boards, and few advantages to speak for. This left it a competition between Arduino and ARM. Arduino was easy to set and use, but had very little real world applications. Additionally Arduino had a large following among hobbyist leading to large samples and libraries, but did not use conventional C, limiting the use of the skills learned. ARM is a very prevalent architecture, with a wide range of boards, but the libraries tend to be restricted by microcontroller. ARM is generally more complex, harder program and set up. In the end we decided that we could overcome the disadvantages of ARM with some hard work on our end, by creating libraries to reduce the program difficulty and having pre-setup projects.

Now that we had an architecture, we had to choose a the board itself. On the recommendation of several professors, we decided to select a board from Olimex, a manufacturer that specializes in making development boards for education purposes. Olimex provided several useful advantages, the first of which was that they sell various hardware

modules that connect to their proprietary UEXT port. Secondly they provide sample code for most of their boards. These two features could save us a large amount of time and effort. We began to examine Olimex's expansive selection of ARM development boards. There were several boards that fit our criteria , but one stood out: the STM32H152, see Figure 3,3.

Figure 3.3 Olimex STM32H152

It has two 2x20 extension ports, a UEXT port, and was only $18. It also had extensive libraries provided, as well as example projects in both eclipse and IAR Embedded Workbench ARM (EWARM). For the full detail, visit:

https://www.olimex.com/Products/ARM/ST/STM32-H152/

This seemed to fit all our criteria, it was simple, expandable, and code libraries that would give a place to start. The STM32H152 uses the STMicroelectronics STM32L152VBT6 microcontroller which is part of the STM32 series of ARM microcontrollers. For the full details For the full product specifications of the STM32L152VBT6 visit:
http://www.st.com/content/st_com/en/products/microcontrollers/stm32-32-bit-arm-cortex-mcus/stm32l1-series/stm32l151-152/stm32l152vb.html?sc=internet/mcu/product/248824.jsp

Unfortunately, this was when we ran into our first, and largest, difficulty. Once the board arrived, we found the libraries provided by Olimex to be either incomplete or flawed. We were forced to find and alternative library, we turned to the manufacturers of the microcontroller,

STMicroelectronics, who had set of libraries for their STM32L1xx line, called the STM32CubeL1. This cube only contained projects and code configured for the development boards built by STMicroelectronics, and their corresponding microcontrollers. Fortunately one of the development board uses a very close microcontroller, the STM32L152C. This allowed us to get our microcontroller booted. Unfortunately the documentation for the libraries was broken, we were forced to puzzle out their construction through trial and error. As we adapted them to match our microcontroller and board. As we explored their construction we made the necessary adaptations to make the libraries functional for our board. After large amount of time, we gained a sufficient understanding the libraries, called the HAL drivers to find and properly adjust them.

3.4 The Modules

In parallel to the choosing and programing of the development board we began to plan how we would expand whichever board we would choose. There were two methods that we discussed, the first was a traditional daughterboard that would contain as much hardware as we could fit. The second was a series of interchangeable modules, based partly on the modules provided by Olimex, and partly on the shields used by Arduino. The daughterboard would be allow us to create a single multipurpose board, it would be complex and would have limited use outside of the kit. The module design would create many single purpose boards, each would have a simple design, but the amount of boards would complicate the projects. We decided to use the Modules because it would allow the kit to be expanded by creating more modules, additionally it would make the kit more versatile, and ,we believe, more interesting to use.

Once we had our our development board chosen, we needed to map out the hardware needed by each project, so that we could determine what was needed on our modules. The full details of this can be seen in Figures 3.4 and 3.5.

Project	Buttons	LEDs	LCD	Speaker	Microphone	Thermistor	Potentiometer
Simon Says +	■	■		■			■
Guitar Hero	■	■	■	■			■
Universal Clock	■		■	■		■	■
Morse Code	■	■	■				
Hangman	■		■				
Voice Recorder	■			■	■		■
Oscilloscope	■		■				
Thermometer			■				
Calculator	■		■				
Maze Game	■		■	■			■
Tilt Maze			■	■			■
Robotic Arm							

Figure 3.4 Project Hardware Part 1

Project	Memory	Voltage Divider	Numberpad	Servo	Motion Detector	Accelerometer
Simon Says +						
Guitar Hero	■					
Universal Clock						
Morse Code						
Hangman	■					
Voice Recorder	■					
Oscilloscope		■				
Thermometer						
Calculator			■			
Maze Game						
Tilt Maze						■
Robotic Arm				■	■	

Figure 3.5 Project Hardware Part 2

We also had to plan how the modules would connect to the development board. Both of the 2x20 ports on the STM32H152 have their own power lines, and connections to 34 Pins on the microcontroller, additionally we had the UEXT port and the Olimex modules. We designed six single purpose modules, to contain the hardware that we would need, see Figures 3.6 and 3.7.

Modules	Buttons	LEDs	LCD	Speaker	Microphone	Thermistor	Potentiometer
Display							
UserI/O							
Sensors							
Servo							
Numberpad							
Soundboard							
Memory							

Figure 3.6 ModuleHardware Part 1

Modules	Memory	Voltage Divider	Numberpad	Servo	Motion Detector	Accelerometer
Display						
UserI/O						
Sensors						
Servo						
Numberpad						
Soundboard						
Memory						

Figure 3.7 Module Hardware Part 2

As we now knew what hardware each project required and what hardware each module contained, we combined the information to see which modules each project would require, see Figure 3.8 for the full details.

Project	Display	UserI/O	Sensors	Servo	Numberpad	Soundboard	Memory
Simon Says +							
Guitar Hero							
Universal Clock							
Morse Code Translator							
Hangman							
Voice Recorder							
Oscilloscope							
Thermometer							
Calculator							
Maze Game							
Tilt Maze							
Robotic Arm							

Figure 3.8 Project Modules

We realized the most commonly needed module for projects was the LCD, so we would use the UEXT port for that purpose, leaving the 2 2x20 ports open for other modules, allowing for the most versatile use of the modules. At this point we chose our LCD Module from Olimex, we choose the MOD-LCD3310, it came with libraries for a microcontroller similar to ours and $8 was inexpensive. For more details visit:

https://www.olimex.com/Products/Modules/LCD/MOD-LCD3310/open-source-hardware

We began to design the modules, we began with the UserIO module as it is the second most commonly use module. It consisted of five multicolored buttons and four multicolored LEDs and their specifically designed to be used with the Simon and Guitar Hero projects, but with the ability to work with the other projects. The second board, dubbed the "sound board", housed a speaker, with the ability to be adjusted by a potentiometer to control the speaker's volume and a microphone and would also be placed on the sound board, since the one project they were needed for also required the speaker. An additional button on the sound board would be used to activate the speaker. The keypad used for the calculator project was placed on it's own separate board. The servo for the motorized arm would also exist on a separate board. The final board was called the sensor board, as it would connect to the thermistor, accelerometer, motion detector, and voltage divider. These were all placed on the same board, since they were each only used for one project and fit a central theme. See Figure 3.9 for some examples of our designing phase

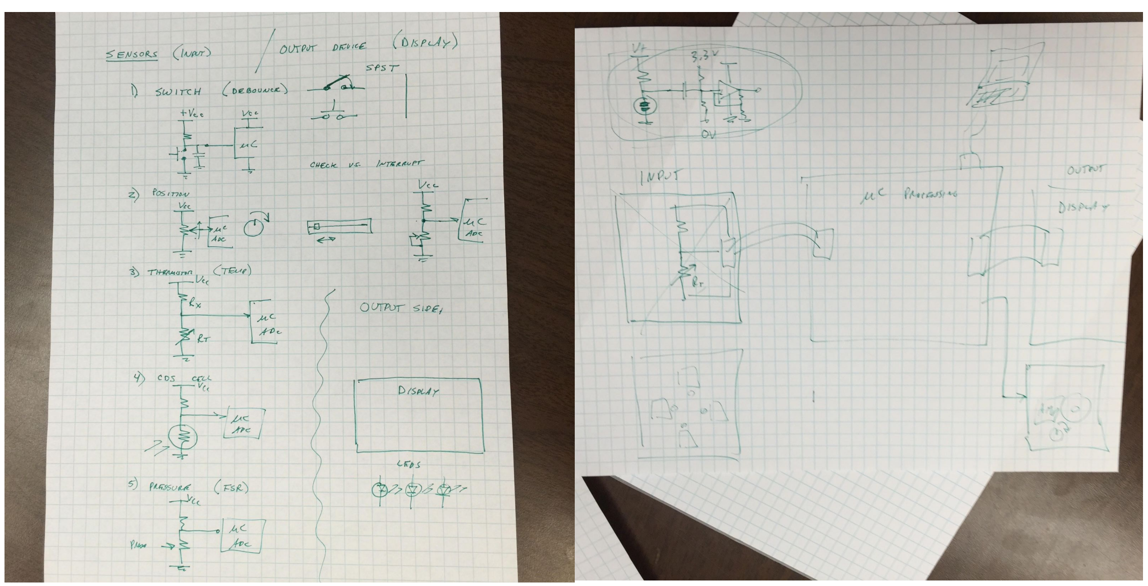


Figure 3.9: Example Design Sketches

When it came to physical construction of the modules we had three options: breadboard, perfboard, and custom PCB. The breadboard was the most easily constructed but was best used only to prototype the boards. Perfboard was an acceptable finished project, and was only a little slower than breadboarding. Custom PCBs were the most professional, but were very time consuming to design and even more time consuming to order. As PCB was the most desireable finish project, we decided to to see what it would take to to design one. We quickly realized that this would take too long, and settled on Perfboard. Figure 3.10 show the various stages of physical development of the UserIO board, as an example.

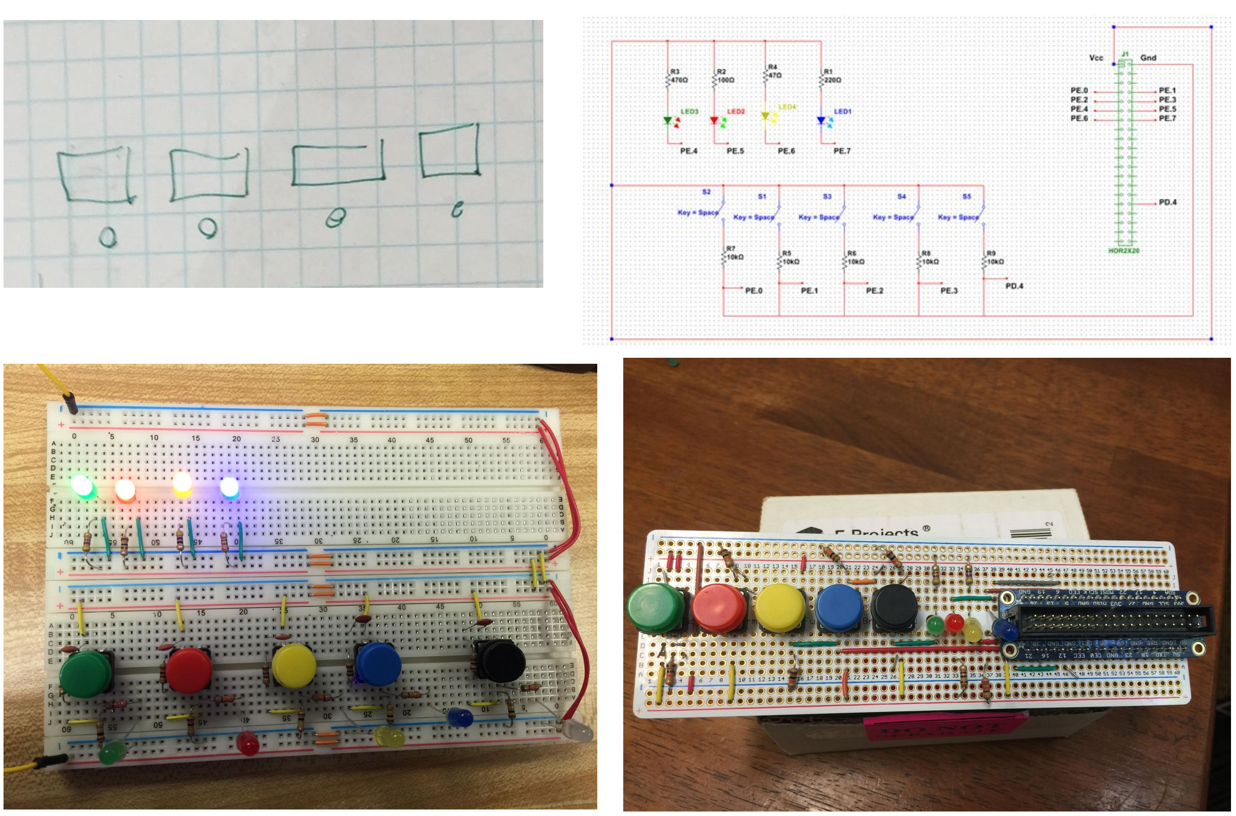


Figure 3.10 Stages of Development

Design, Circuit, Prototype, Product

In addition to the physical construction of the modules, additional software functions needed to be developed to initialize the modules and allow the microprocessor to properly connect and interface with the daughterboards. The LCD screen was distributed by Olimex. The files containing the code needed to initialize the LCD, as well as basic functions used for writing and drawing on the LCD, could be found on the Olimex website. The files, once downloaded, required a few alterations. First, the main file needed to include a link to the.h file for the LCD. Second, the GPIO pins required to initialize the LCD needed to be changed to correspond with the pins that connect to the microcontroller via the UEXT.

For the modules that we constructed, such as the keypad module, we created drivers that would initialize and interact with them. To continue with the keypad as an example, the driver contained to functions: keypadInit() and getKey(). The function keypadInit() would initialize the

four GPIO pins connected to the columns as Push-Pull output pins, and the four pins connected to the rows are initialized as input. The function “getKey”, individually set the GPIO pin for one of the columns as high, and then polls each of the pins in the row to determine if they were pressed, before resetting the GPIO pin of that column and then setting the next column, and then polling each of the pins in that row. If one of the buttons has been pressed, then the function returns an integer value associated with the button that was pressed.

It was at this time we ran into our second major difficulty, one that resulted in some drastic changes in both our project and pace. We tried to design all of these modules, while setting up the board, ignoring the lessons. We were trying to do too much with too little time, so we narrowed what we working on. With just the keypad module functioning as a buttons and the LCD, we could build several of the projects. We began to focus on these, and the UserIO module which was almost complete at the time and those projects. This would allow us to created the lessons for those projects, and have complete sections, hardware, software, and Lessons.

3.5 Writing and Testing the Lessons

The final part of our project was writing and testing the lessons. As we completed the modules needed by each lesson we began writing and testing the lesson. By the time we got around to writing up a lesson plan, we still weren’t sure how the lessons would be implemented to properly educate the student to build the application. So, we worked backwards: we designed the final project, and then built an application that could successfully meet the parameters that we set for the project. There were multiple advantages to this method. First, it proved that the kit contained all of the necessary components needed to fully implement the application. Secondly, it gave us a better understanding of those components, as we found limitations, bugs and other such problems. Thirdly it allowed us to break down the steps that we took to build the project,

and as well as any challenges or difficulties we had. This information was immensely useful when writing the lessons as it allowed us to decide of the order of the lessons and the what assignments each lesson needed to teach student how to build the applications. We decided at this point to add an introductory lesson, one that would teach the students how to connect the development board to their computer, how to use the IDE EWARM, and how to use the basics of the libraries provided. The results of these are discussed in the results section.

4 Results

This section will cover the outcomes of the project both the deliverables and the difficulties of the project. There were three parts to the deliverables hardware, software and lesson plans. When the project first started we had twelve lessons and seven daughter modules planned, but we quickly learned that this was very over ambitious and our timeline was based on an underestimation of the difficulties before us. In the end we delivered six lessons and only three of the modules. The modules we delivered were the LCD, the Keypad, and the User I/O. The LCD module was bought from Olimex, the manufacturer of our development board. An image of it can be seen in Figure 4.1.

Figure 4.1: Olimex MOD-LCD3310

The keypad module, seen below, was designed and constructed by us, it is a prototype of what would be in the final kit where we would like it to be on PCB. The prototype is constructed on a perfboard due to time constraints, it is a simple design of pull down resistors for the column pins. The circuit diagram for this module can be seen in Appendix G.

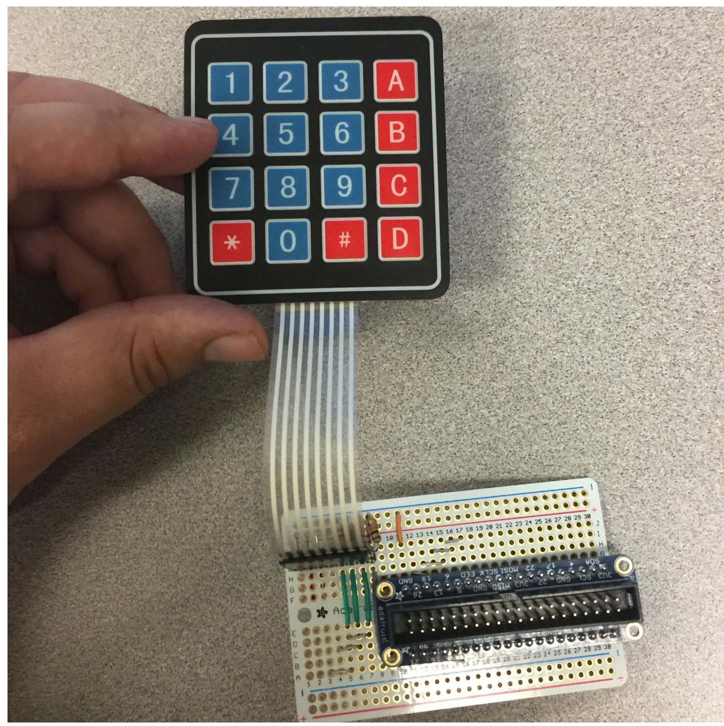


Figure 4.2: Keypad Module

The User I/O module, seen below, was designed and constructed by us, it is a prototype of what would be in the final kit where we would like it to be on PCB. The prototype is constructed on a perfboard due to time constraints. The module consists of five additional user push buttons and four additional LEDs and their accompanying circuitry. The circuit diagram for this module can be seen in Appendix H.

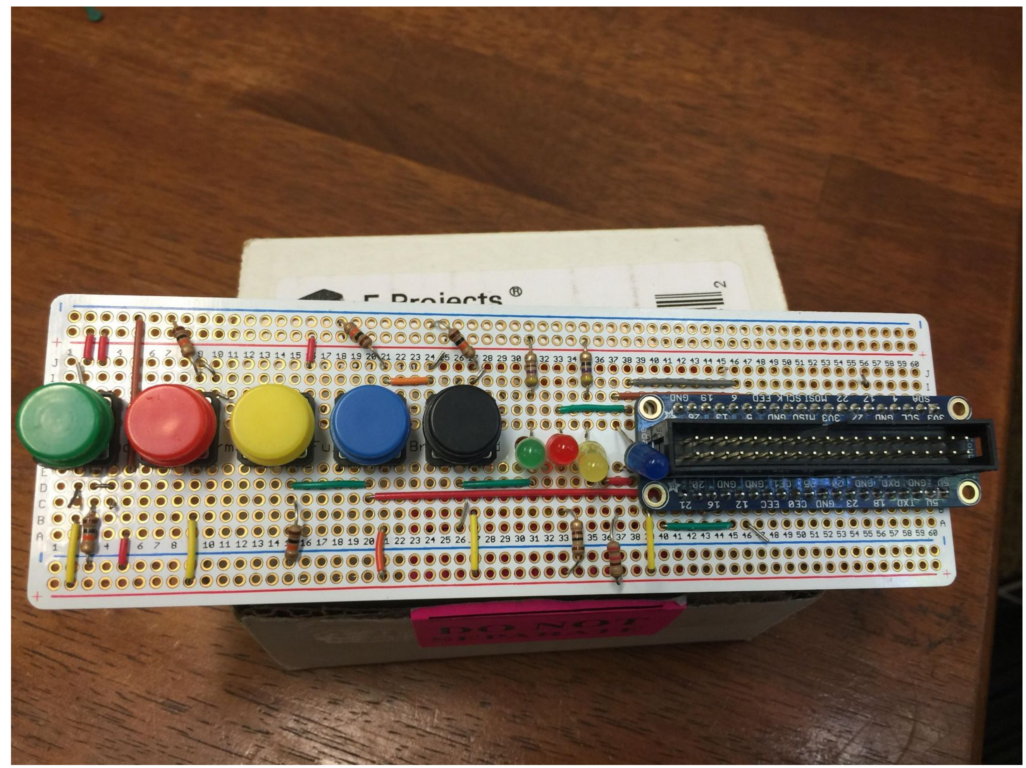


Figure 4.3: User I/O Module

The lessons are the Introduction, the Calculator, the Clock, the Morse Code Interpreter, the Maze Game, and Simon Says. Each lesson is broken into assignments, each a part of the lesson that ends with the student making an application that will teach them one topic or element to make the final assignment, the goal of each lesson work. The full lessons can be seen in Appendices A-F, what follows is a brief description of each lesson. The introduction is shows the students how to set up their kit, and start using the software. It teaches how to make a very basic application on the development board and how to initialize pins. In the calculator lesson students will learn how to use the Keypad and LCD daughterboards, in order to create a functioning calculator. They will do this by learning step by step how to write to the initialize and write to the LCD using the drivers provided in the kit. Then they will likewise learn to initialize and read input from the keypad. Once they have learned these assignments they will combine them to create a calculator.

The clock lesson will teach students how to use their kits to build a clock. It is meant to teach them how the HAL Library keeps time and how to access and interpret that information. This project will explore how to read through the HAL Library, and make minor adjustments as well a introduce the concepts of timers. It will culminate with them creating an accurate clock that can be set. The morse code lesson is meant to introduce student to the concept of interrupts, and why they are a useful. It will also teach Morse Code, but that might be less useful. The Maze Game lesson, teaches students how to use the drawing functions built into the LCD to design a working maze game. This project will also help reinforce the concept of programming dynamic states within a code. In the final lesson we completed, Simon Says, the student will learn to create their own basic driver for the module and then use their own drivers to make the game of simon says.

5 Conclusions

This section will discuss what we learned over the course of this project and what what we think could be done to take the kit we have started further.

The first and most important lesson we learned from this project was to not try to do to much, especially at one time. We were trying to develop multiple modules and lessons while still setting up the board. Trying to do all that resulted in getting almost none of it done. After a while, and with some advice from our advisor, Professor Bitar, we narrowed our project, taking it step by step, trying to get one lesson and its components working. We then optimized what we were working on focusing on the lessons that used, or could be adapted to use, the same components from the first lesson. We learned the difficulty and time it can take to develop your own ideas into reality, and that we needed to plan better for obstacles. We also learned new engineering concepts as well, we learned to how to construct PCBs, though we decided that it was to time consuming to use. We set up drivers for our students, and enhanced our knowledge of the subject by creating lesson to teach students.

We did not get everything we wanted to have in the kit complete, as such we have ideas on how this our kit could be developed further. First we have the four modules we did not have time to build: Sound, Memory, Sensor, and Servo. Appendices I-K contain charts showing our original plans. The sound module was meant to contain speaker and a microphone, which would serve to teach how to use ADC's and DAC's as well as sampling. The memory module was to contain small EEPROM storage device, which could be used teach about memory management. The sensor module which was to contain three sensor: a thermistor, a accelerometer, and a motion detector, which was meant teach how to interpret sensor data. The final module that was proposed was the Servo, which was meant to introduce the the idea using an embedded to system to control a physical response. The unfinished project were meant to use these modules to teach the lessons.

In conclusion, while there is much that could be done to further our kit, we have gotten the initial lessons and modules completed. In doing so we learned how to better plan projects,

work around obstacles, and to prioritize our work. We also learned to work as a group, and to divide tasks, to get them done as quickly and efficiently as possible. This project has made us into better engineers, readier to face the challenges of life and work that we will now go on to have.

Works Cited

1. Nerdkits, LLC, “Nerdkits: Electronics Education For a Digital Generataion”, 2013. http://www.nerdkits.com/. 6/27/2016.
2. Olimex, “Olimex: Development Boards”, 2016. http://www.olimex.com. 5/1/2016.
3. Adafruit, “Netduino Go! Starter Pack (Modular .NET microcontroller)”. https://www.adafruit.com/product/800. 6/27/2016.
4. Maker Shed, “Make: Getting Started with the BeagleBone Black Kit - Version 2”, 2016. http://www.makershed.com/products/make-getting-started-with-the-beaglebone-black-kit . 6/27/2016.
5. Microchip, “PICDEM Lab Development Kit”, 2016. http://www.microchip.com/Developmenttools/ProductDetails.aspx?PartNO=DM163045. . 6/27/2016.
6. Homce, Ellie and Homce, Don. “Microcontroller”, 2007. http://www.parents-choice.org/product.cfm?product_id=22036&. 6/27/2016.
7. Meeting Tomorrow. “History of the Microprocessor” . https://meetingtomorrow.com/content-library/history-of-the-microprocessor. 6/28/2016.
8. Singer, Graham. “The History of the Microprocessor and the Personal Computer” 9/17/2014. http://www.techspot.com/article/874-history-of-the-personal-computer/ 6/28/2016.
9. Robot Platform. “History of Microcontroller”, 2016. http://www.robotplatform.com/electronics/microcontroller/microcontroller.html. 6/28/2016.
10. John. “Microcontroller – Invention History and Story Behind the Scenes”, 8/23/2013. http://www.circuitstoday.com/microcontroller-invention-history. 6/28/2016.

11. Copeland, B. Jack, "The Modern History of Computing", The Stanford Encyclopedia of Philosophy (Fall 2008 Edition), Edward N. Zalta (ed.), <http://plato.stanford.edu/archives/fall2008/entries/computing-history/>.
12. Bestofmedia Team. “Computer History 101: The Development Of The PC”, 8/ 24/2011 http://www.tomshardware.com/reviews/upgrade-repair-pc,3000.html. 6/28/2016.
13. Zandbergen, Paul. “Central Processing Unit (CPU): Parts, Definition & Function - Video & Lesson Transcript ” http://study.com/academy/lesson/central-processing-unit-cpu-parts-definition-function.html. 6/28/2016.

Appendix A - Introductory Lesson

Lesson 1 How to use your Kit

This Lesson will teach you how use the kit you have purchased. It will discuss what is included in the kit and how to set it up. You will then write a some very basic applications that will help you take your first step in embedded programing. To use this kit, you will require a USB compatible computer with Windows 7 or newer. You should also have at least a basic understanding of the C programming language. We are sad to say that this lesson is a bit tedious, it was to write as well, but the lessons that follow will be much more focused on doing is learning.

Need to Know 1 What's in Your Kit

Welcome to your Introduction to Embedded Systems Kit, inside your kit you will find six items. Seen below in Figure 1.

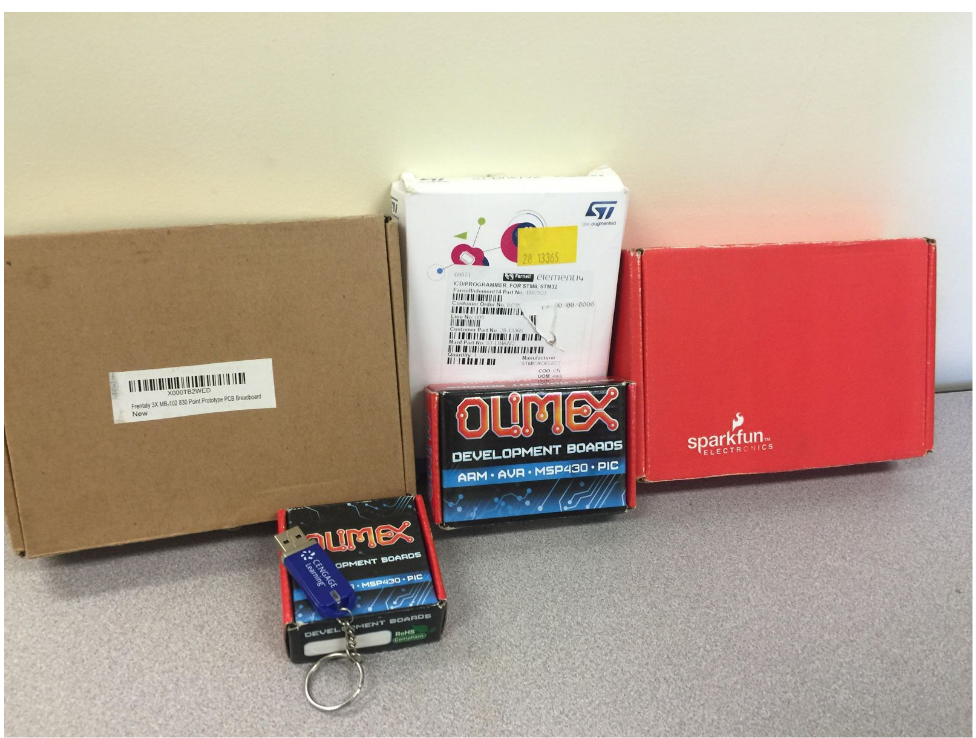


Figure 1: Embedded Systems Kit

The first item, which you have already figured out if you are reading this, is a flash drive. This drive contains two folders and README.txt, which directs you to this Lesson. The first folder is Lessons, which contains the Lessons you will be completing while using this kit. The second folder is STM32H152 Kit, this folder contains all of the code, libraries, and projects you will need to use this kit. I would recommend that begin using this kit by transferring the contents of the drive to your computer for ease of use.

The second item in the kit is the a small rectangular box labeled Olimex Development Boards, which contains the Olimex STM32H152, pictured in Figure 2.

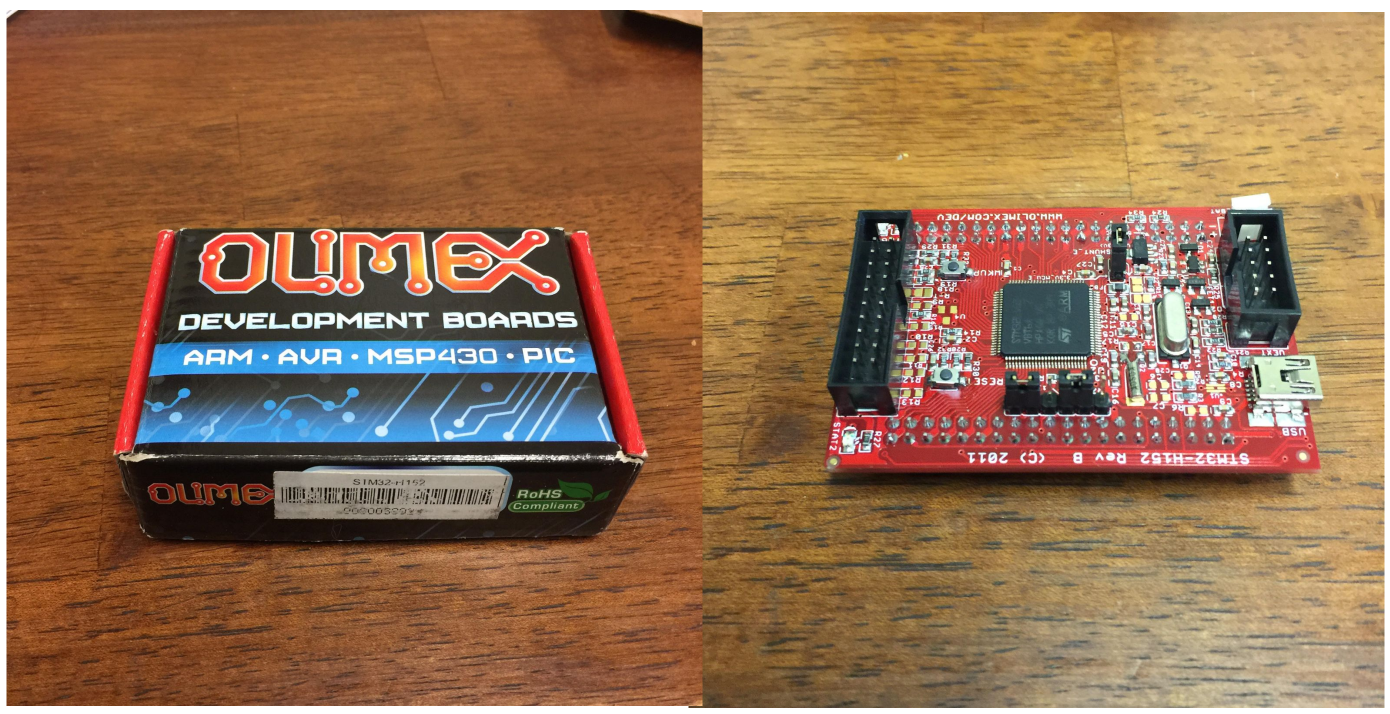


Figure 2: Olimex STM32H152

This is the board that contain the your new microcontroller, and will run the applications (hardware and software) you create throughout your lessons.

The third item in your kit is a box labeled STMicroelectronics ST-Link/v2 which contains three items: a MiniUSB to USB cable, a 2x10 Ribbon Cable, and the STLink/v2, seen in Figure 3.

Figure 3:STMicroelectronics STLink/v2

This item connects the development board to your PC, allowing you to program and debug your applications.

The next three items in the kit are the modules that connect to the development board. These modules are the LCD, the Keypad, and the UserIO. Each one adds functionality to board. The LCD Module is the Olimex MOD-LCD3310, it and the 2x5 Ribbon Cable are in a small square Olimex box both can be seen in Figure 4.

Figure .4: STMicroelectronics STLink/v2

The second module, and fifth part of your kit is the Keypad Module which is contained in red square box in your kit, along with the2x20 ribbon cable needed to connect it, seen in Figure 5.

Figure 5: Keypad Module

The final part of your kit is the UserIO Module, this module contains a series of button and LEDs and is contained in the brown box, along with the2x20 ribbon cable and second MiniUSB to USB cable, seen in Figure 1.6.

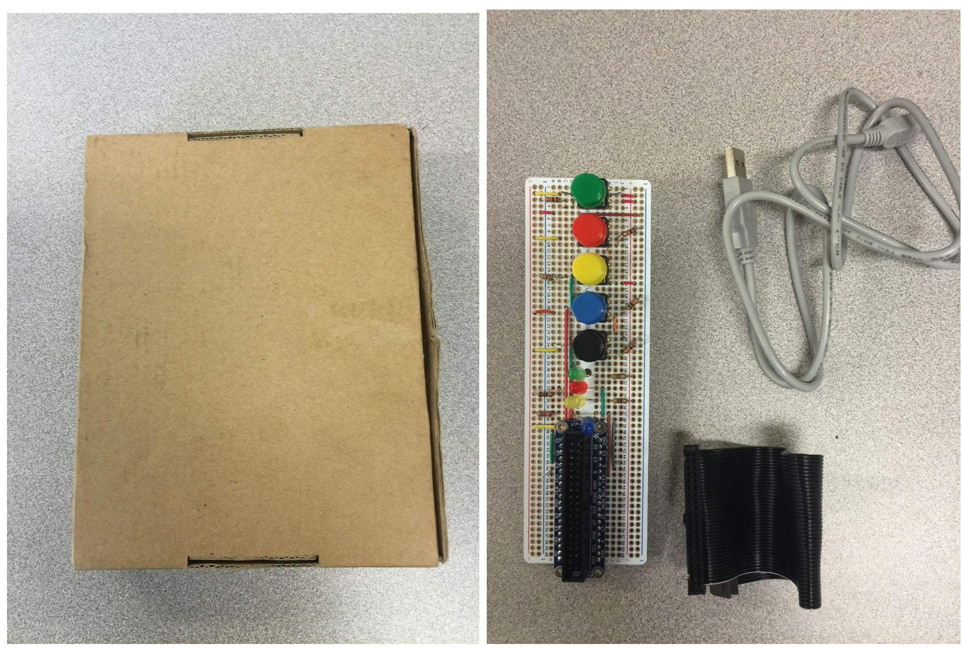


Figure.6: User IO Module

Need to Know 2 Kit Structure

Now that you know what is in your kit, you need to know what how this kit works. The Kit has three components: hardware, software, and Lessons. All the hardware you'll need is included with the kit. There are three main components of the hardware, the first of which is the development board, which will often be referred to as the Board, see Figure 2. This kit uses the Olimex STM32H152 as its Board, for the full technical specifications (specs) you can visit https://www.olimex.com/Products/ARM/ST/STM32-H152/ . This board is the center of the kit, it contains the microprocessor, a STM32L152VB, for its specs visit http://www.st.com/content/st_com/en/products/microcontrollers/stm32-32-bit-arm-cortex-mcus/stm32l1-series/stm32l151-152/stm32l152vb.html?sc=internet/mcu/product/248824.jsp .It will be what is running your applications. The second part of the hardware are the Modules, these are

boards that connect to the Board and expand it functionality. There are three modules, seen above in Figures 4-6, are named after the functionality they add: LCD adds a graphical display, Keypad adds a 4x4 Keypad, and UserIO adds buttons and LEDs. The final part of the hardware are the connectors the wires and cables used to connect the Board, the Modules, and your PC. The most important of these is the STLink/v2, which allows your PC to directly interface with the microcontroller.

The second component of the kit is the software. Included in the kit are the drivers for the Board, two out of three of the modules (Lesson 6 is creating the third), and the Microcontroller itself. All of the software is included in the Folder STM32H152 Kit, seen in Figure 7.

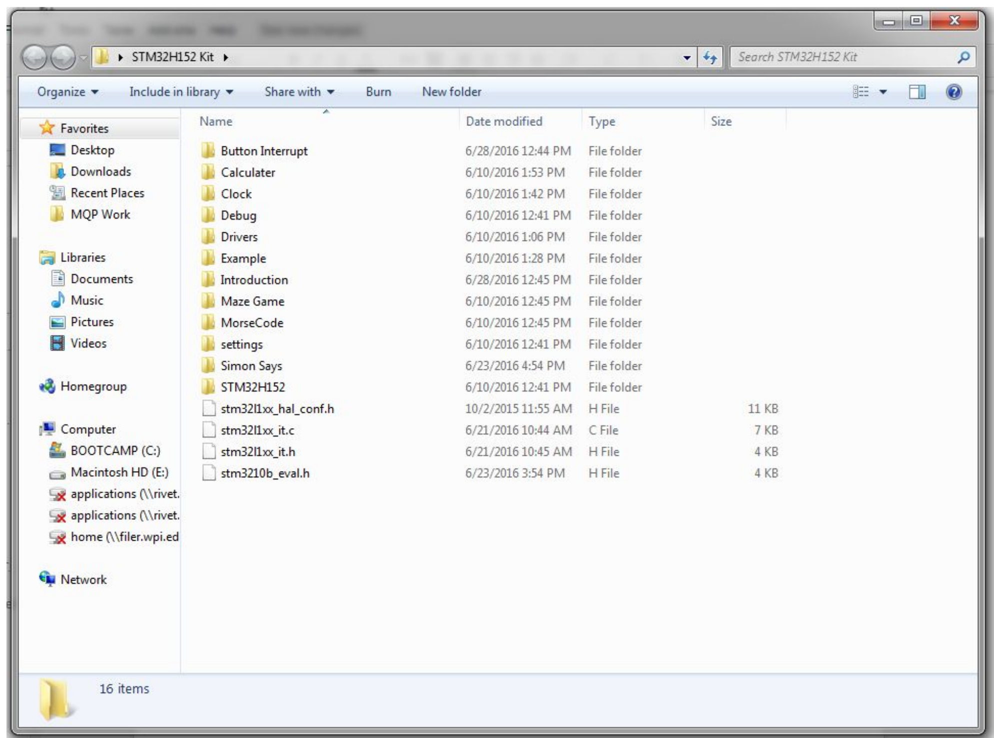


Figure 7: STM32H152 Kit

The drivers folder contains all of the files used to run the microcontroller, which will discussed in more depth as you complete your Lessons. Most of the other folders contain the Workspaces for use in IAR embedded workbench, the IDE that this kit will teach in. They are

pre-setup to help you jump right into creating applications. I will use the one named Example as, well, an example it can which be seen in Figure 8.

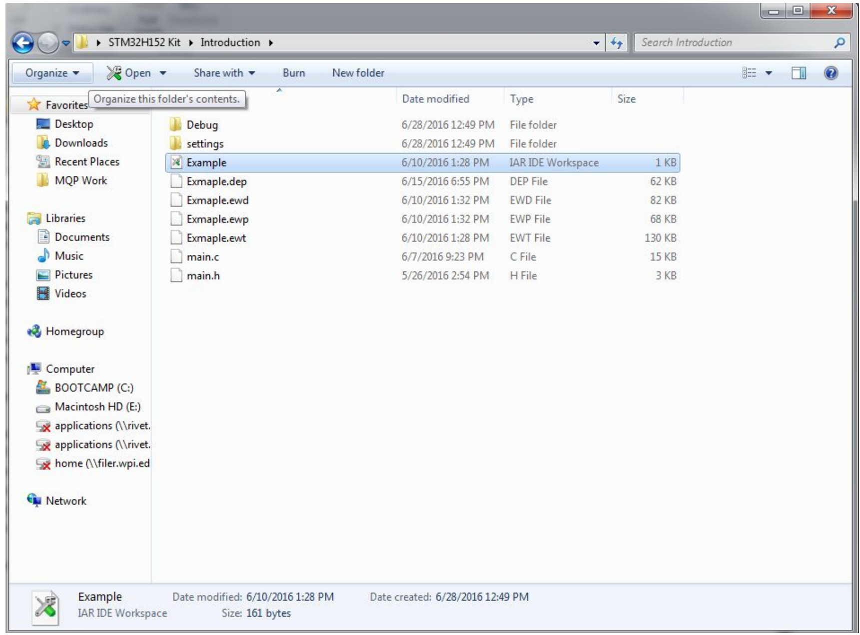


Figure 8: Example Folder

The important file here is the Example.eww (show as just Example) which is the file that will contains the workspace for your Lesson. The only piece of software not included in the kit is the IDE you will using, IAR Systems Embedded Workbench: ARM Edition (called EWARM for short). An explanation of how to acquire a copy is in Need to Know 4 of this Lesson.

The final component of your kit is the Lessons, a series of worksheets that will teach you the basic elements of embedded programming. Each lesson is split into parts each with a different purpose. The Need to Knows are purely informative sections, they have no tasks or activities to complete. We tried to avoid these as much as possible as we find learning is easier and more fun with hands on work. The Assignments are the section you will find the most. Assignments are meant to teach you by making you build and run applications. Each Lesson, save this one, will culminate in a final Project. The Projects are applications that will require everything you have learned in that Lesson, and its predecessors. The applications are a

calculator, a multi-function clock, a morse code interpreter, a maze game, and Simon Says. Note as you progress through the Lessons, we will be providing less and less help.

Need to Know 3 Hardware Set-Up

Now you know what your kit contains and how it is organized, but how do you set-up your kit? This section will teach you how to connect all those wires. To begin with let's hook your Board to your PC. First get your STLink/v2 and its cables, seen in Figure 3, and connect them as seen in Figure 9.

Figure 9: STLink/v2 Set-Up

From there connect the 2x10 pin ribbon cable's other end to the matching plug on the Board, see Figure 10 for more details.

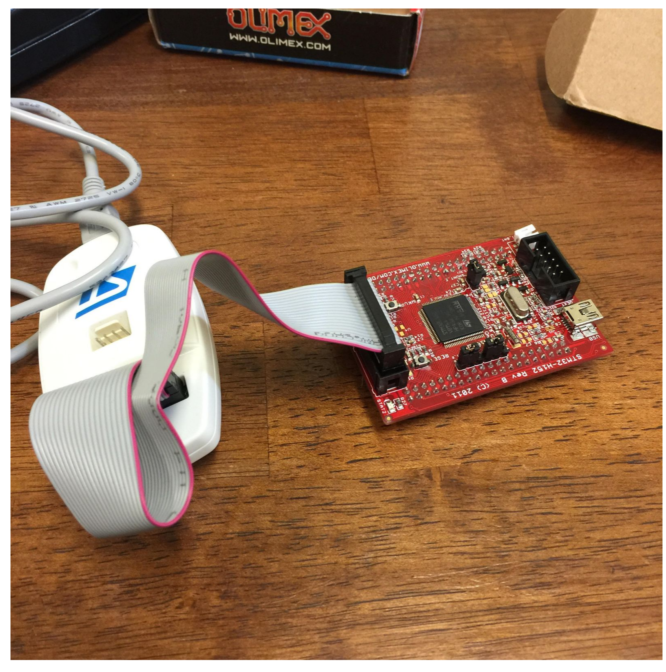


Figure 10: STLink/v2 Connected to STM32H152

Finally plug the USB cable into your computer. The finished product can be seen in Figure 11. You will need to have this set up for every Lesson, as this is the minimum you will need to run an application.

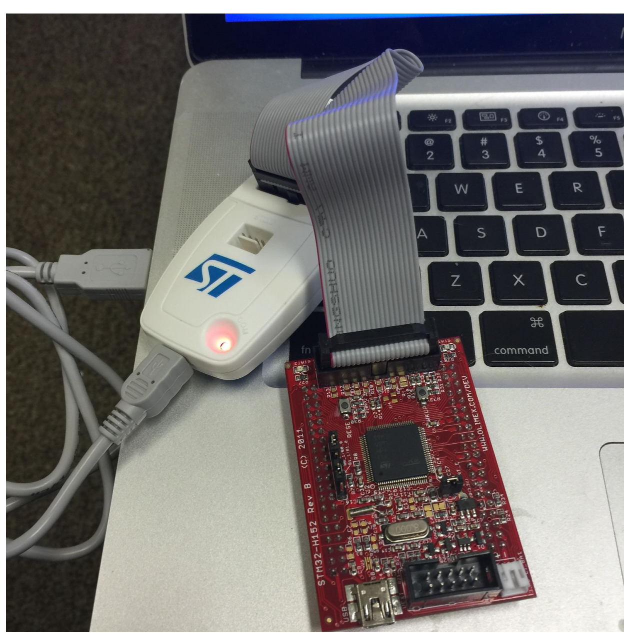


Figure 11: Board is Ready to Go

I will also explain how to connect the modules to board. These are not necessary for every Lesson, connect them now to test it out, but only connect them when needed after. The LCD Module is unique and will use a separate port that the other two modules. The LCD Modules box contains the 2x5 ribbon cable used to connect the Module to the Board, see Figure 4. Connect the cable to the back of the LCD, be certain the tab lines up to the hole. Connect the other end to other open, matching port on the Board, see Figure 12.

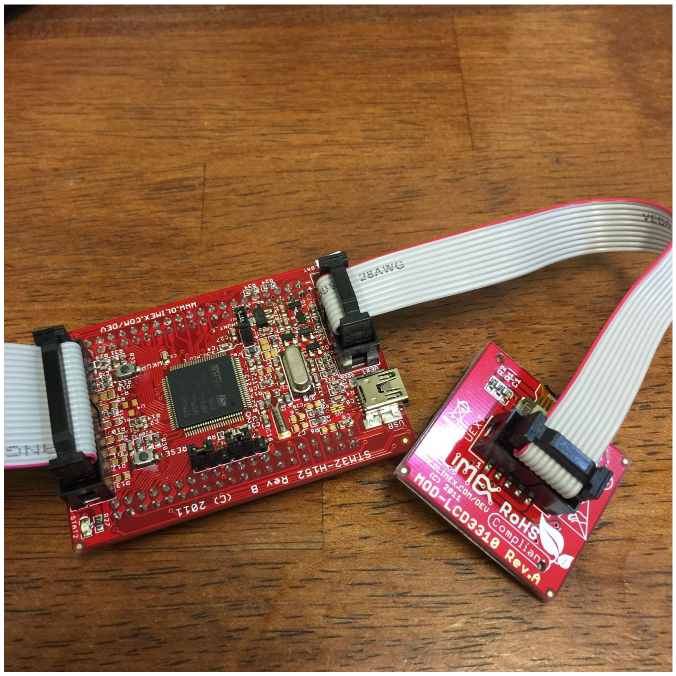


Figure 12: LCD Hooked Up

The Keypad and the UserIO are not used simultaneously in any lesson, and use the same port, Con1 on the back of the Board. These boards are connected with one of the 2x20 pin ribbon cables included with them. It is important to note that Con1 does NOT have a tabbed connector, place the tab the Tab so that it faces INTO the Board, see Figure 13, I mean it.

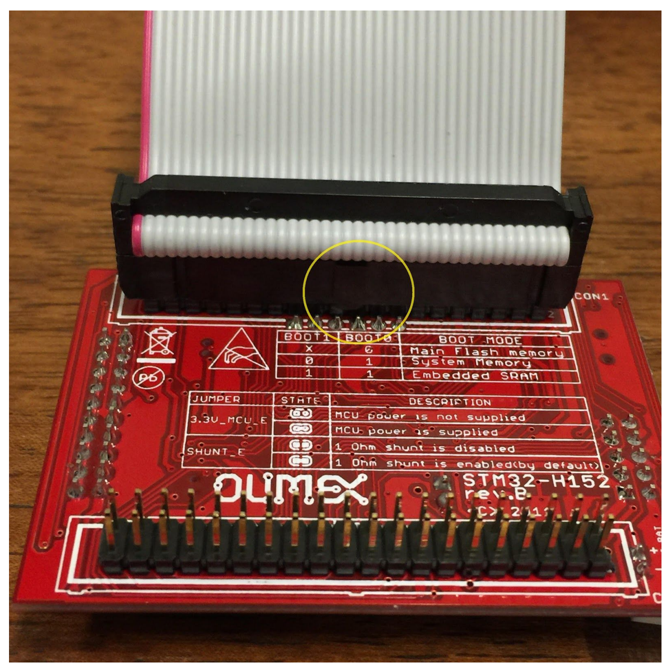


Figure 13:Con1 Hook-Ups

Figure 14 contains the full hook-up of the Keypad Module..

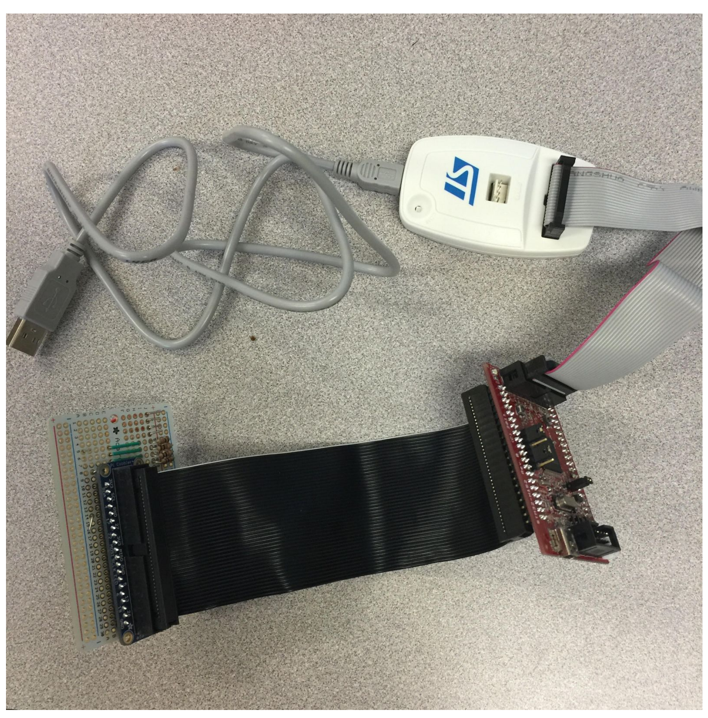


Figure 14:Keypad Module Connected

At his time I need to note that the STM32H152 was originally designed to work with the original STLink not the STLink\v2. There will be no problems with programing or debugging, the major change was that Vcc (Power) went from 5V to 3.3V, as such the power converter built into the Board, regulate the power to ~2.2V, not 3.3V. This can be easily corrected by using the extra MiniUSB to USB cable provided, connect the USB end to a power source (your PC, your phone charger, etc.) and connect the MiniUSB end to MiniUSB port on your Board. For the full connection of the UserIO Module, see Figure 15

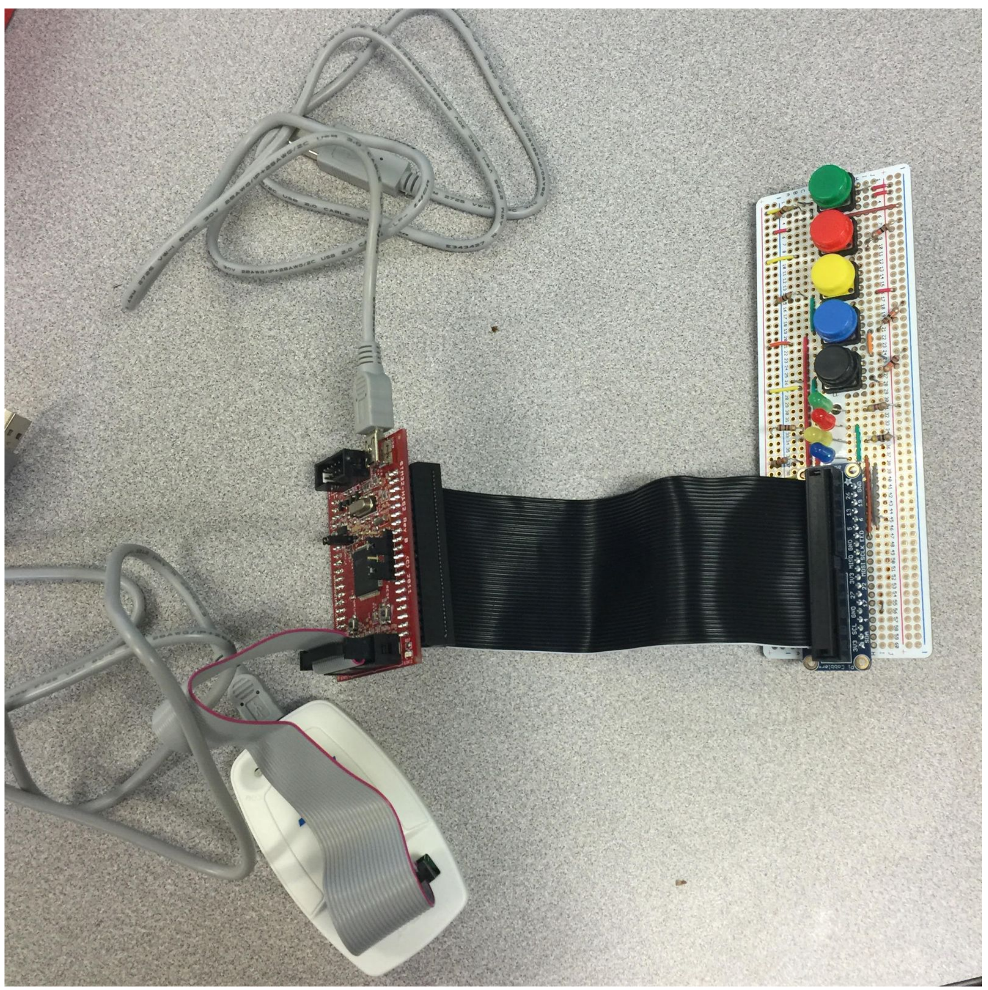


Figure 15:UserIO Module Connected

Need to Know 4 Software Set-Up

This section will teach you how to set up the software needed for the kit. Luckily for you, the setup and configuration of the workspaces has been done for you. This means that the only thing you need to set up is the IDE. Go to https://www.iar.com/iar-embedded-workbench/#!?architecture=ARM¤tTab=overview

Pictured below in Figure 16.

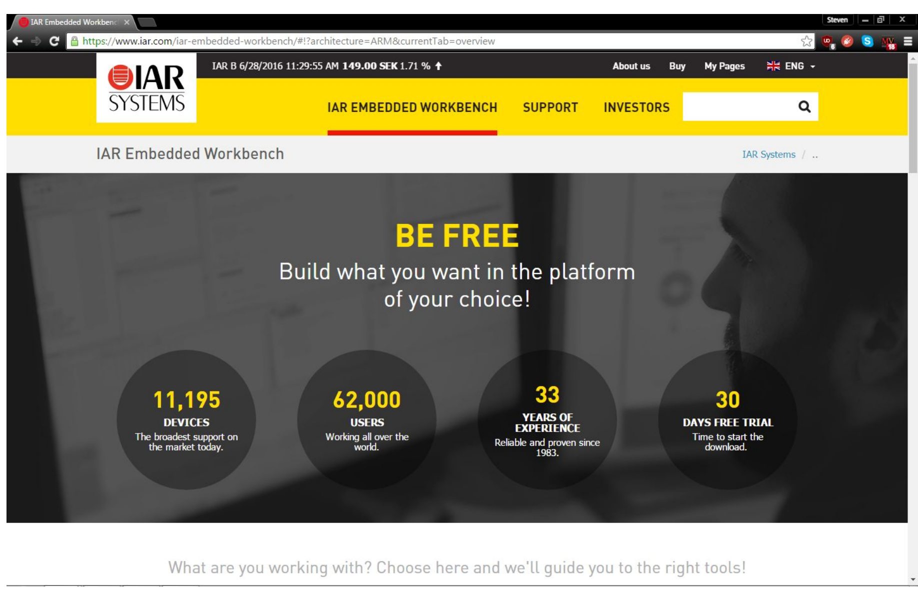


Figure 16:IAR EWARM website

Scroll down until you see Figure 17.

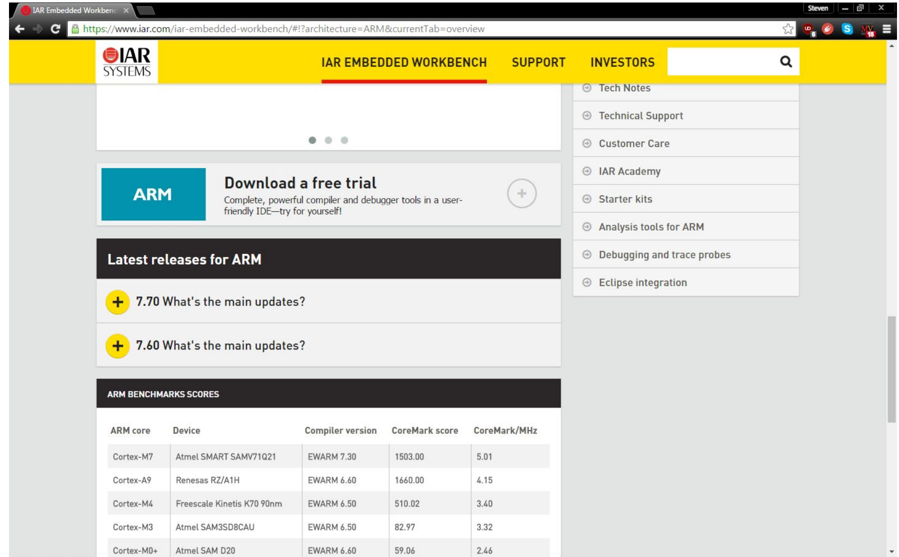


Figure 17:IAR EWARM Download Location

Expand the download a free trial and click the yellow download button. This will download the file *EWARM-CD-7701-11486.exe*. Execute the file. Follow the setup and registration guide provided. When registering choose the kickstarter version, otherwise you only get 30 days.

Go to the example folder from Figure 8. You'll note that inside this folder, contains two additional folders, labeled "Debug" and "settings", an IAR Project Workspace with the name "Example", four additional files associated with "Exmaple", and both a .c file and .h file for "main". Every folder that is associated with one of the assignments contains an IAR Workspace with the same name as the lesson. The workspace is where you'll be writing and debugging your code for each lesson. Double click on "Example" to open up the project within IAR Embedded Workbench, see Figure 18.

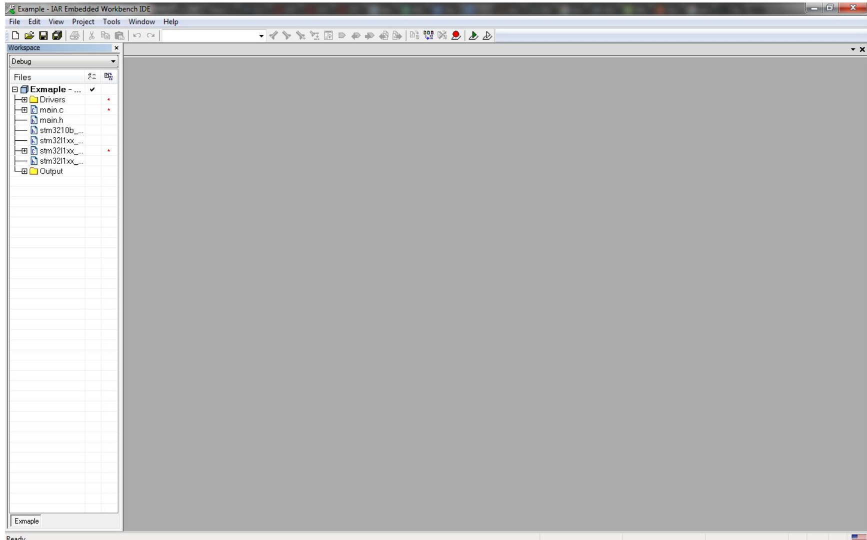


Figure 18: Opened Workspace in EWARM

Assignment 1 The First- Blink an LED

Now it is time for you to start doing things go to the Introduction Folder, and open the example workspace contained. Once you've opened Workbench, draw your attention to the left side of the window. There should be a section labeled "Workspace". All of the files that are used in the lesson are listed here. Open the file "main.c". This is where you'll be writing your code for your first assignment, see Figure 19.

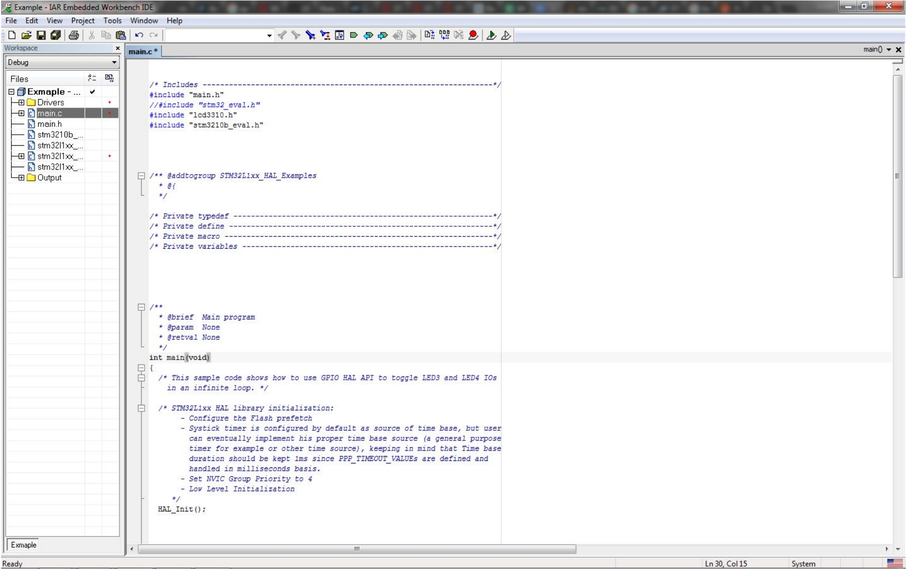


Figure 19:Ready to Start

This first assignment is very simple, as the main focus is getting you used to the structure of the lessons for future assignments. You will learn how to initialize a red LED and have it blink continuously. Note you will only need the Board for this Assignment.

Within your main function, you'll notice that there are already several lines of code. This code is used to initialize and set the IO pin that you will be using for this lesson. In a standard main function for an embedded system, the code is divided into two sections. The first section contains code that only needs to be run at the start of the application, including variable declarations, and initialization and configuration of hardware, see Figure 20.

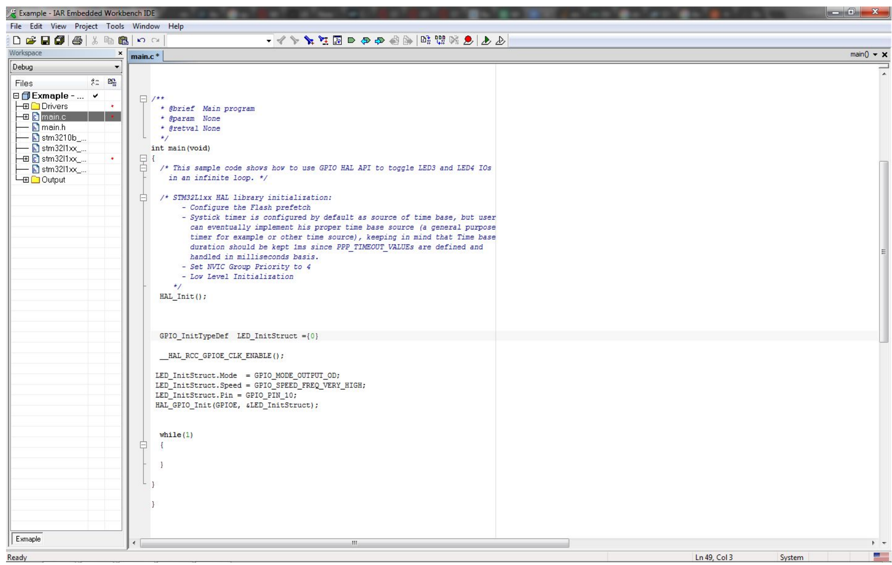


Figure 20: Main Function

HAL_Init();

- This function is used to initialize the HAL library.

GPIO_InitTypeDef LED_InitStruct ={0}

- This creates a struct containing the information required to initialize a pin as a GPIO (stands for General Purpose Input Output. GPIO is a standard digital logic pin).Each pin on a microcontroller has multiple purposes, depending on how they are initialized.

__HAL_RCC_GPIOE_CLK_ENABLE();

- This function enables the GPIO Clock, at port E.

LED_InitStruct.Mode = GPIO_MODE_OUTPUT_OD;

- This sets the mode that the GPIO pin fulfills; in the case of an LED, this is an output. There are two types of Outputs: Open Drain(OD) or Push-Pull(PP). In this case we’ll be using OD.

LED_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;

➢ This sets how often the pin's current state is refreshed,

```
LED_InitStruct.Pin = GPIO_PIN_10;
```

➢ This sets what pin is being called. The red LED is connected to pin E10.

```
HAL_GPIO_Init(GPIOE, &LED_InitStruct);
```

➢ This fully initializes the pin according to the parameters set forth.

The second section of the code is the actual function of the application. It is often placed within a while(1) loop to ensure that it runs continuously. The while(1) loop that appears in the application is at the moment completely empty, see Figure 20.

The function HAL_GPIO_TogglePin will toggle the specific pin that is called, and alternate between setting the pin high or low. You can also use HAL_Delay to set a small delay between two commands. The code below will flash the red LED on, and then off:

```
HAL_GPIO_TogglePin(GPIOE, GPIO_PIN_10);
HAL_Delay(100);
HAL_GPIO_TogglePin(GPIOE, GPIO_PIN_10);
HAL_Delay(100);
```

Add this code to while(1) loop as seen in Figure 21.

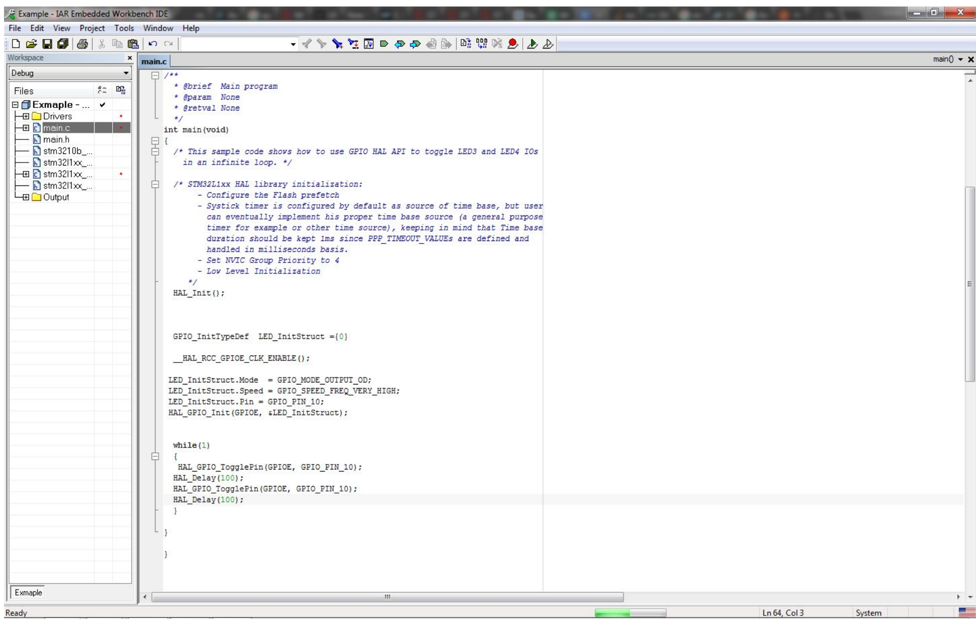


Figure 21: Completed Main Function

Once you have finished with your code, you will need to download it to the board and debug the code. Find "Download and debug", which can be found under the project tab,(or alternatively, by using the shortcut Ctrl+D). If you have successfully written the code without any errors, the screen will change to debugging mode. To start the debugging process, press the green arrow, and the red LED should start blinking.

Assignment 2 Building a Switch

For your second assignment, you will learn how to initialize the button on the microcontroller, and then use the button to toggle the LED on and off.

Initializing the button is very similar to initializing the LED, with a few exceptions. First, the button on the microcontroller is located at pin A0, and the port should be registered as an input pin rather than an output. Insert the code below into your application below the initialization of the LED, see Figure 22:

```
GPIO_InitTypeDef Buttoninitstruct = {0};
```

```
__HAL_RCC_GPIOA_CLK_ENABLE();
Buttoninitstruct.Mode = GPIO_MODE_INPUT;
Buttoninitstruct.Speed= GPIO_SPEED_FREQ_HIGH;
Buttoninitstruct.Pin = GPIO_PIN_0;
HAL_GPIO_Init(GPIOA, & Buttoninitstruct);
```

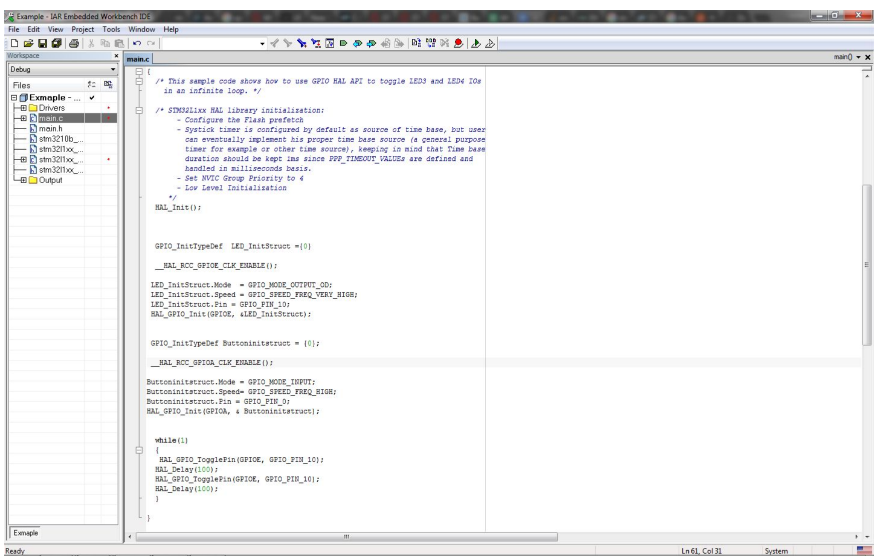


Figure 22: Button Initilized

Within your while(1) loop, you'll need to design a method to toggle the LED every time the button is pressed. The function HAL_GPIO_ReadPin can be used to read in the value of a specific pin. The following code is an if statement that will wait for the button to be pressed, and will toggle the LED on and off. Replace the code you added above and enter the following in its place, see Figure 23:

```
if ( HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0) == GPIO_PIN_SET){
        HAL_GPIO_TogglePin(GPIOE, GPIO_PIN_10);
        HAL_Delay(100);
}
```

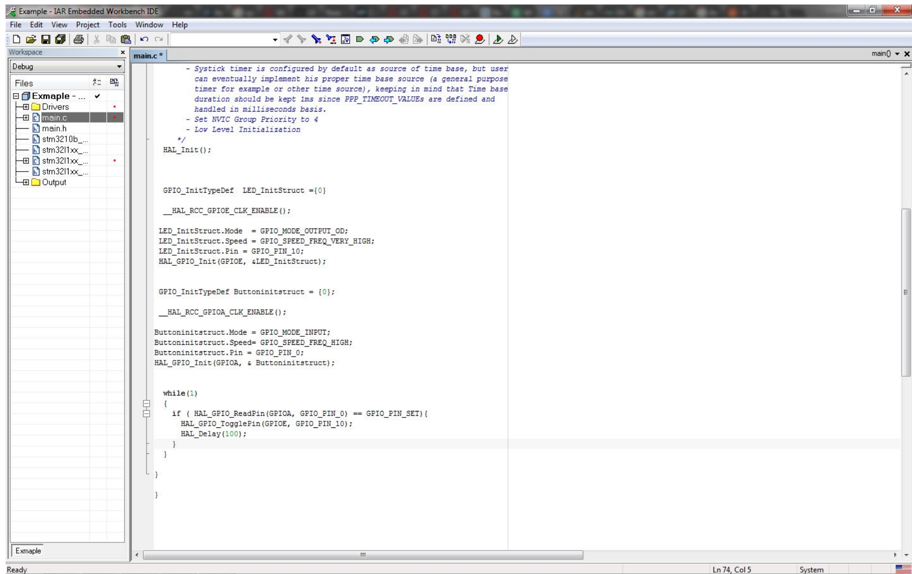


Figure 22: Butten Toggled LED

The delay is necessary for this code to work, as the processor moves at incredibly fast speeds. Without the delay, the LED will rapidly toggle on and off several times while the button is pressed down, which will lead to inconsistent results when pressing the button.

These assignments hopefully helped you understand the principle behind initializing pins on the microprocessor board.

Appendix B Calculator Lesson

Lesson 2- Make a Calculator

In this lesson, you will learn how to use the Keypad and LCD daughterboards, in order to create a functioning calculator. You will learn step by step how to initialize and write to the LCD using the drivers provided in the kit. You will likewise learn to initialize and read input from the keypad. Once you understand these two concepts you can combine them to make yourself a calculator.

Before you begin, you'll want to attach the LCD Screen and Keypad to your Microcontroller, as you'll be using both of these modules for your assignments. The LCD screen comes packaged with a UEXT connector cord to link to the main board. On the top of the microcontroller board is a 2x5 row of pins within a black enclosure. Connect one end of the UEXT cord to this point, and connect the other end to the back of the LCD Screen. Keep in mind that the LCD should be oriented so the side of the screen with the black bar is on the top.

Next, the Keypad board. On the bottom side of the Microcontroller board is two 2x20 rows of pins. They are labelled on the board as "CON1" and "CON2". These pin rows will directly connect to the daughter boards, which contain a 2x20 row of receiving pin holes.. Connect alongside CON1, with the indent tab facing downward (into the board). Refer to Lesson 1 incase you forgot.

Assignment 1 - “Hello World”

This very simple phrase, Hello World, is commonly known amongst programmers as the first message that people learn how to send. In this first Lesson, you will be sending your own “Hello World” Message.

Open the folder in STM32H152 Kit labeled “Calculator”. Inside your main file is the pre-written code, see Figure 1:

```
LCDInit();
LCDContrast(0x70);
int Inverse =0;
```

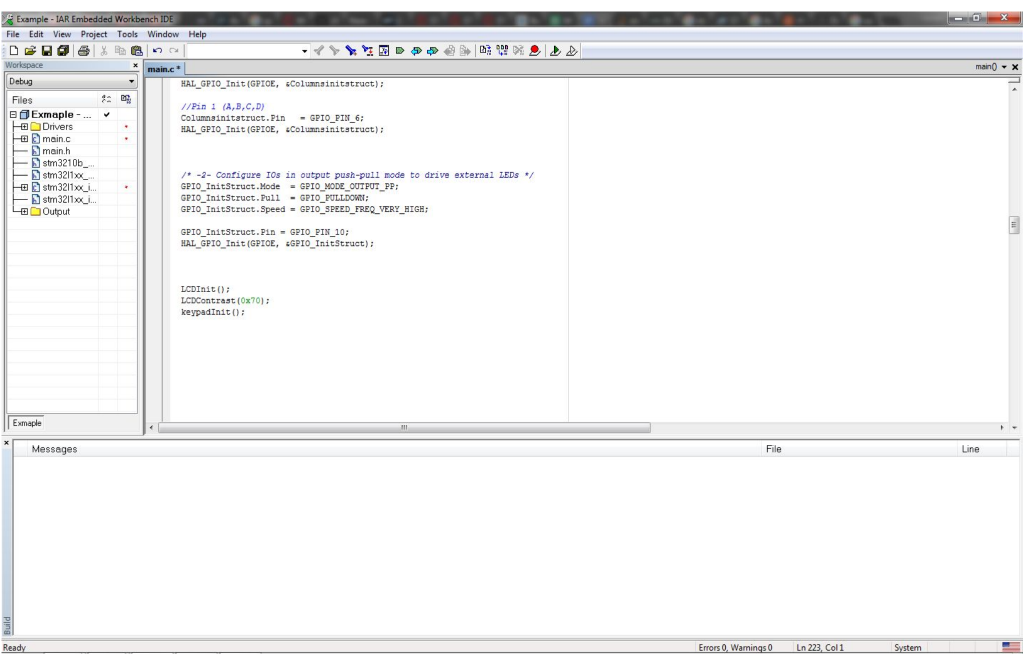


Figure 1:LCD Initialization

LCDInit will properly initialize the pins used to interface with the LCD screen, and then reset and clear the contents of the screen. LCDContrast will set the standard for the color contrast between the blank screen and the text that appears0. This will become important later.

You’ll be using the function LCDStr to write to the screen. LCDStr can take a character string, and display the string on the LCD screen across a certain row, starting from the left and displaying all the way to the right. LCDStr takes in three parameters: the first is an integer that sets the row that the message appears on, the second is the string of chars that makes up the message, and the third parameter sets whether the message appears as black text on a clear background, or as white text on a black background. Place this line of code in your while loop, see Figure 2:

```
LCDStr(0, (unsigned char *) "  Hello World ", Inverse);
```

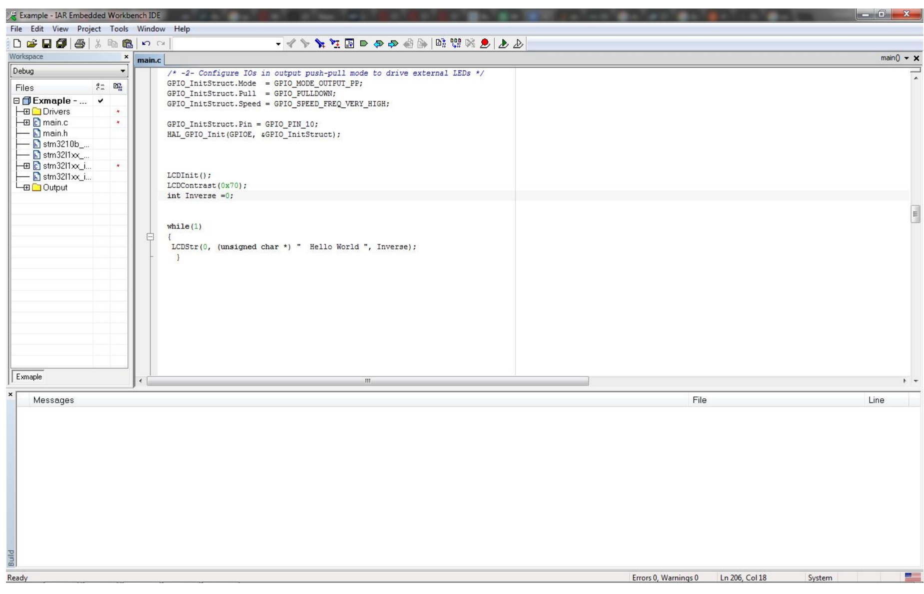


Figure 2:Hello World

Build and Download your code onto the microprocessor board. You should see the text “Hello World” displayed on the top of the LCD screen, in what is roughly the center.

Assignment 2- Errors with LCDStr

Now, replace your current code with the following, see Figure 3

:

LCDStr(0, (unsigned char *) "We are going to the moon!", Inverse);

LCDStr(1, (unsigned char *) "Yay!", Inverse);

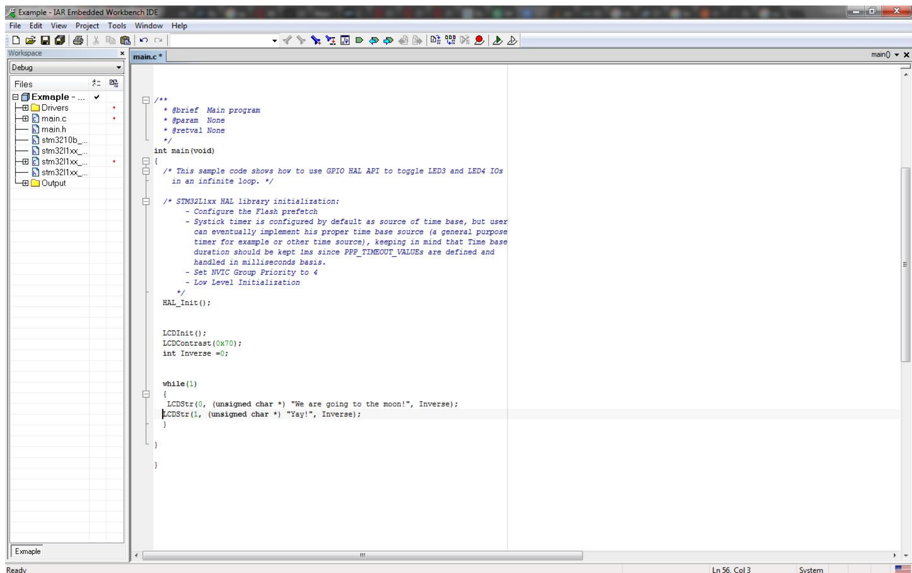


Figure 3:LCD Going to the Moon

If these two snippets of code ran as intended, the LCD screen would display two messages on two lines: “We are going to the moon!” on one line, and “Yay!” on the second line

Now when you run your application, you should see Figure 4.

Figure 4:Yay!e moon!

Your assignment is to figure out why this error occurred, and then to repair it. (Hint: The LCDs only so big)

Assignment 3- Function intToString

LCDStr can only write a string of chars onto the board, and so far this has not resulted in a problem. However, in the coming lessons it might be necessary to also display individual numbers on the screen as well. Included within the example functions is a handy piece of code called “intToString”, which takes in a char array and an integer, and then fills the array with the integer in string form, see Figure 5.

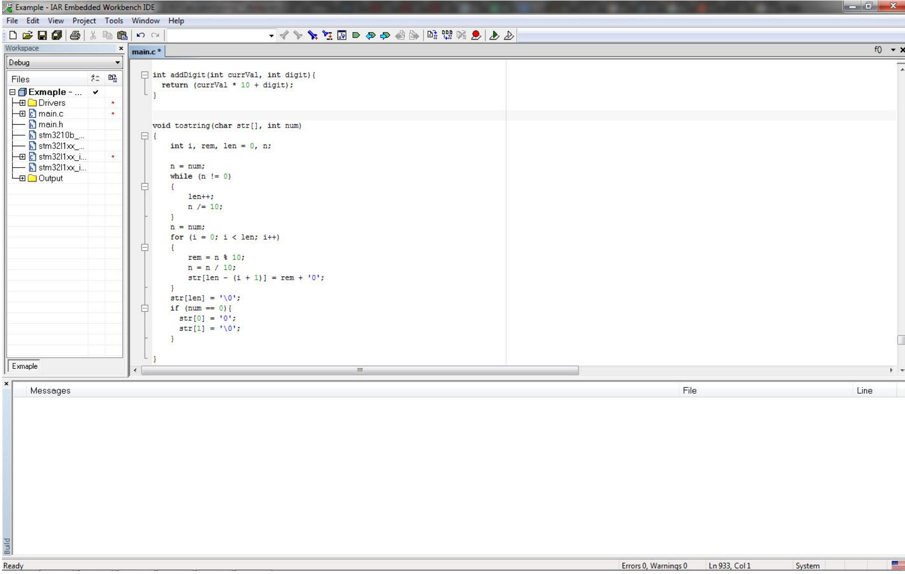


Figure 5: intToString()

We'll provide a basic explanation how it works:

> First, the length of the integer is determined by setting a while loop that continuously divides the integer by 10, and incrementing a variable "len" by 1 for every tens place. Once the number of tens places is determined, the function creates a For loop that takes each integer place and transforms the last place number into a char. The char is then placed within the string, and the integer shifts to the right. When all of the decimal places within the integer have been entered into the string, the character '\0' is placed in the array. This character is very important as it marks where the array ends, and any empty values in the array that appear after the \0 are not addressed. This prevents a potential error of a function trying to read an empty string.

To test how it works, create a variable "result" and define it as an int. Within the main function, create a mathematical equation (for instance, 16*2, or 4*3-5/2) and set the variable to

be equivalent to the given result. Then, using the function intToString, turn the result into a string of chars, and display the result on the LCD screen.

Need to Know 1 - Keypad

The next board you'll be acquainted with is the Keypad board. Included in the modules folder in driver is the driver file Keypad.c, see Figure 6.

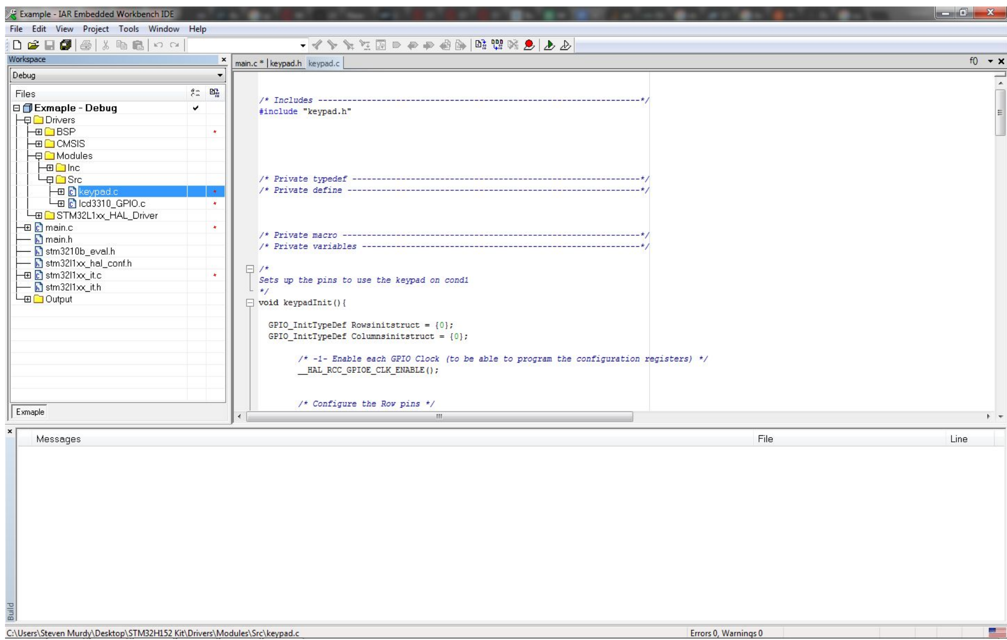


Figure 6: Keypad.c

This driver consists of two functions, keypadInit(), which will initialize the Keypad Module and getKey() which will return a number based on which key is pushed when called, see Figure 7 for details.

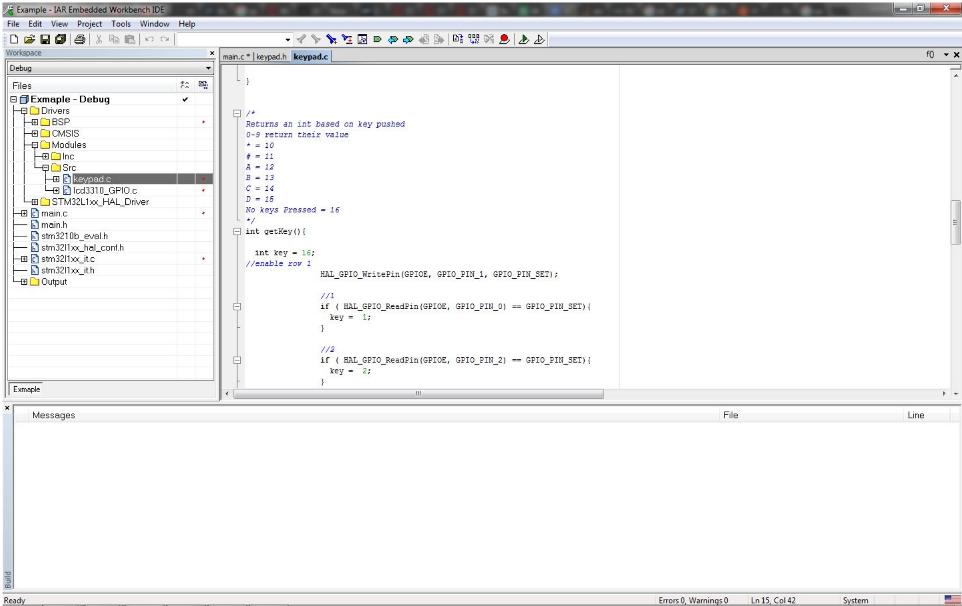


Figure 7: getKey()

Assignment 4 Wait for a Keypress

You'll need to have a method of calling the function getKey, and also record the result. Create an int variable, and then assign the variable the equation getKey, like in the code shown below:

```
int key=16;
key=getKey();
```

This will assign the variable "key" the value of whatever button was pressed when getKey was called. Create a function that will continuously wait for a key press, and then display the message "Hello" if it reads a successful key press.

Assignment 5: Messager

In the previous assignment, any key press would cause a message to be displayed. Using that code as a base, create a new function that will display "Hello" if the 1 key is pressed, and displays "Goodbye" if the 2 key is pressed. (Eventually, you will need a code that will perform different tasks for every type of key pressed, but for now, you should limit yourself to just 2 buttons.)

Project 1 Calculator

Assignment: This is your first fully-realized project, and you'll have to finish it without direct assistance. We can't step you through how to do this! You'll have to use the skills you recently learned to build a calculator. Your calculator, once completed, should meet the following criteria,

- ❏ Number keys, when pressed, must display the correct number.
- ❏ Screen should display the first number, the second number, and the result
- ❏ There should be a button for Adding, Subtracting, Multiplying and Dividing
    - ❏ Negative Numbers
- ❏ The operation keys must work properly
- ❏ Screen must display a string of numbers at least 4 characters long
- ❏ You must have a method of clearing the screen with a button press

There are also a selection of bonus challenges that you can use to personalize your application. These are mostly things you can add for fun and they may require more thought and planning to finish.

- ❏ Display all three numbers (the two operands and the result) on the screen at the same time
- ❏ Create a visual representation of what number has been pressed

- ❏ Configure your clear function so it erases the number being entered first, then the entire screen if pressed again.
- ❏ Display results of 8 characters
- ❏ Clear screen after every equation
- ❏ For the division equation, display the result as an integer with remainder
- ❏ Have Equations transition into each other (2+2=4+2=6-3=3). (This will be the hardest bonus requirement to accomplish.)

(Hint: If you're not sure exactly where to start on this lesson, we have provided one piece of example code that initiates the 0, 1, 2, and * key, and can write basic addition equations that are one character long. You may use this code as a reference.)

Additional Help: It's a good idea to constantly comment your code as you write. This will help you keep your thoughts organized and allow you to easily parse through your code should you come across an error that you can't find

Appendix C Clock Lesson

Lesson 3 Telling Time

This lesson will teach you how to use your microcontroller to build a clock. To do this you must understand how the HAL Library keeps time and how to access and interpret that information. This lesson will explore how to read through the HAL Library, and make minor adjustments as well a introduce the concepts of timers. This project will require the LCD and Keypad Modules.

Assignment 1 Seconds Run Timer

The first step to build a clock on a microcontroller is to know the speed of the internal clock. Now the question is how to learn this begin by opening the example workspace in the Clock folder. . Go to function HAL_Init(); in the code and right click. Select Go To Definition "HAL_Init()", see Figure 1, this should bring you to the file stm32l1xx_hal.c, at the location of the definition of HAL_Init().

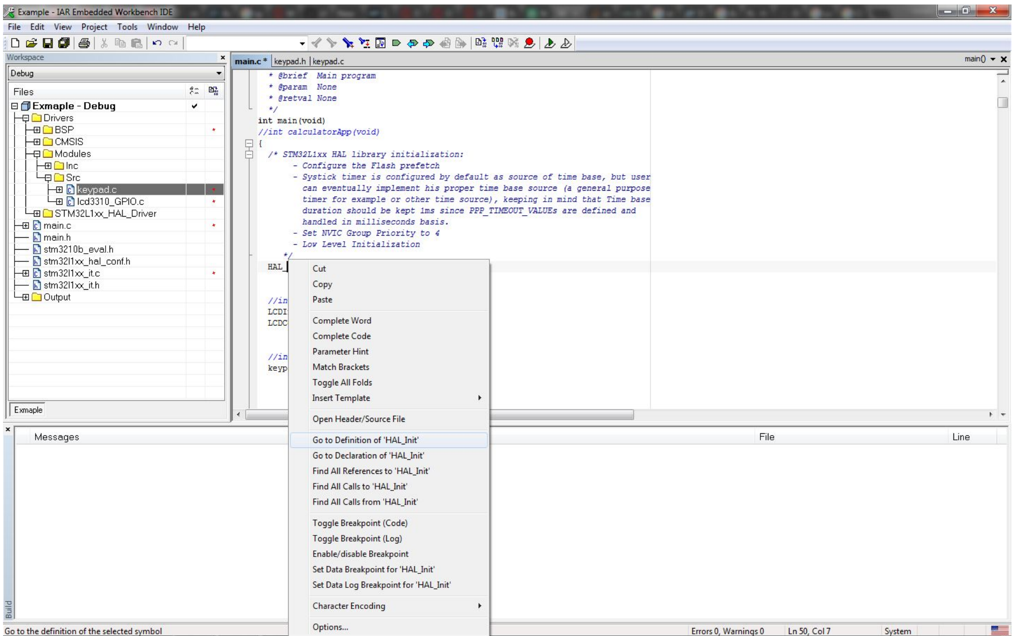


Figure 1: Go to Definition

In the definition there a call the function HAL_InitTick(), this function initializes the clock for the microcontroller. Repeat the Go to Definition except on HAL_InitTick(). This should take you to the definition of the function. The comments tell you the base time that the system keeps track of a useful number if you want to build a clock. No figure for you, I want you to find this yourself.

Go to the private variables section of stm32l1xx_hal.c, here you will see the declaration, see Figure 2.

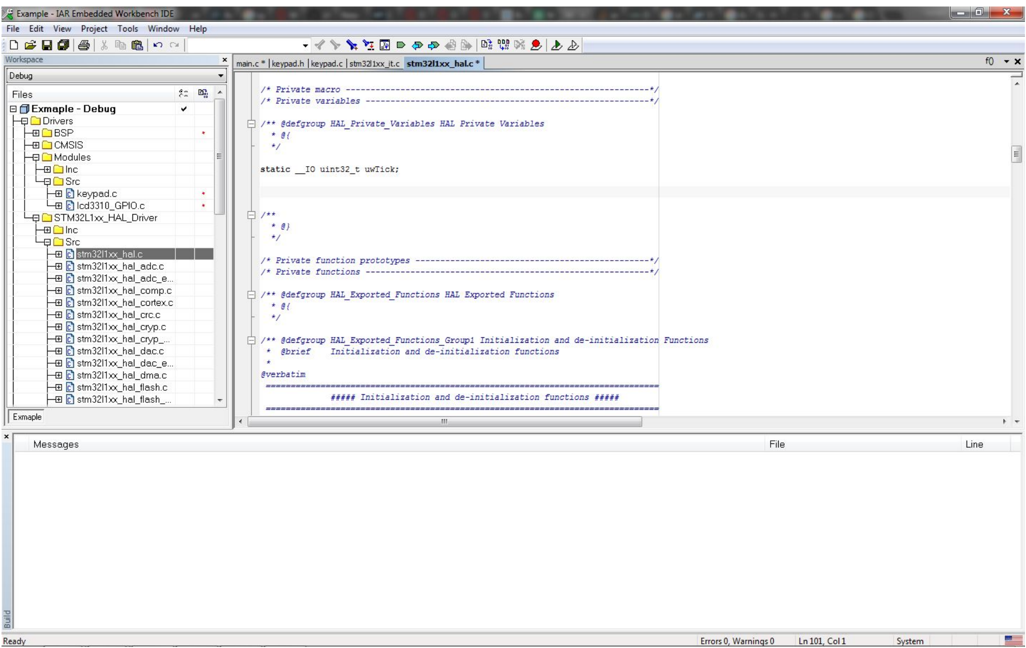


Figure 2: Private Variables

uwTick is the variable that the HAL libraries stores its clock. If you look at where it is used in the code you will see the functions:

```
__weak void HAL_IncTick(void)
__weak uint32_t HAL_GetTick(void)
```

These two functions are how the system clock works. HAL_IncTick is called as an interrupt by the system as it gets input from the hardware clock built into the microcontroller. HAL_GetTick is how you get the time.

Now that you have the knowledge it is time for you to to build the first part of your clock the seconds. Add the code:

```
int seconds = 0;
```

Underneath the declaration for uwTck in the private variables section, see Figure 2. Next go to the function HAL_IncTick and add a line of code that increments the variable seconds every second. (HINT: Use the base time of uwTick you learned earlier and the % function.)

Now you need to add a function based on HAL_GetTick to give you access to seconds in your main function, call it getSeconds. To access your new function you will need to prototype it in your header file, stm32l1xx_hal.h, see Figure 3 for is location in the Workspace map on the left side of the screen.

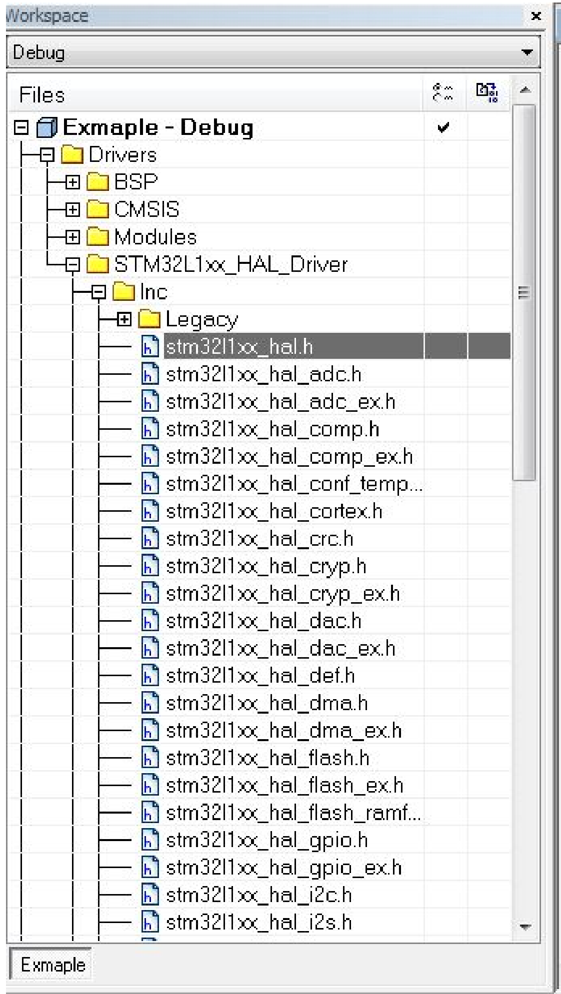


Figure 2: Private Variables

Add the prototype under HAL_GetTick. Now return to your main.c, add the following code to the initialization part of the main function:

```
char secondString[14];
```

Now add the following code to the while(1) loop, see Figure 4:

```
 toString(secondString, getSeconds());
  LCDStr(0, (unsigned char *) secondString, Inverse);
```

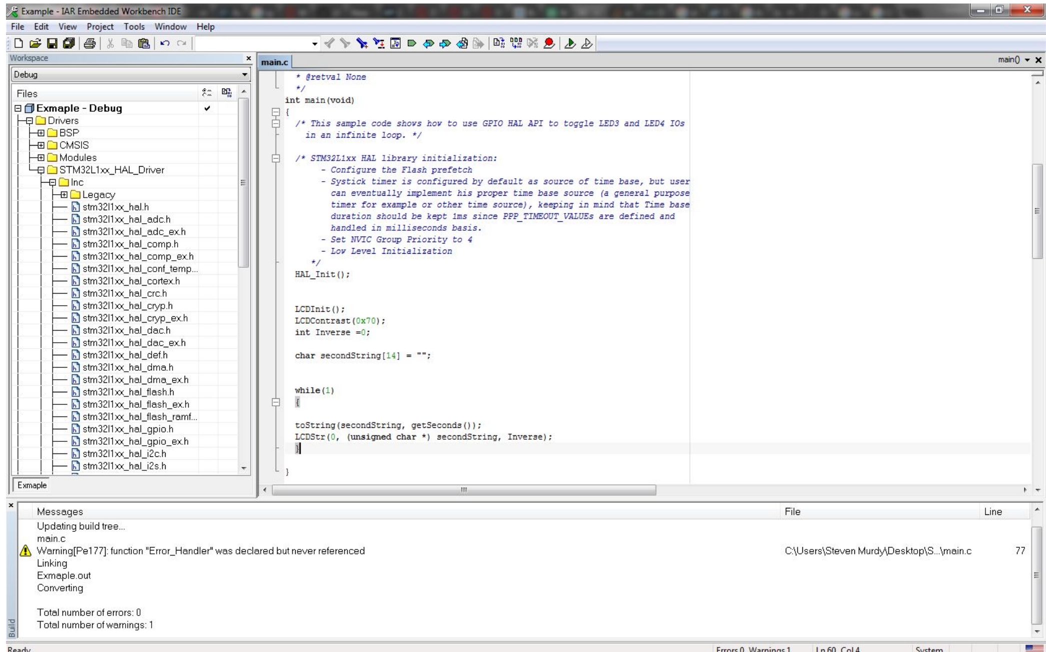


Figure 3: Seconds Timer Code

This code should display the seconds your program has been running (to 14 digits), see Figure 4 for what the result should resemble. NOTE: The function intToString has been included in the starting code for this example to assist you.

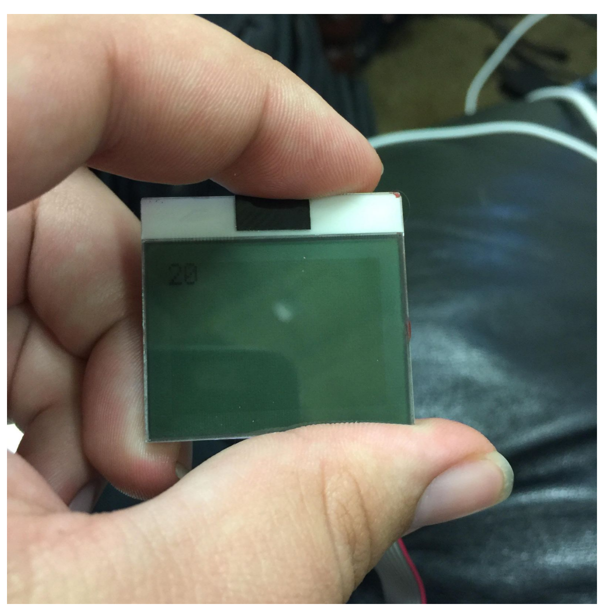


Figure 4: Seconds Timer

Assignment 2 HMS Run Timer

This assignment will take you a step further, you will now make a timer that does Hours: Minutes:Seconds. Start from where you left Assignment 1, go to stm32l1xx_hal.c and add the following private variables under seconds.

```
int minutes = 0;
int hours = 0;
```

Now add the get functions getMinutes and getHours, don't forget to add the prototypes in the header file. Next you will need to edit HAL_IncTick to roll seconds into minutes and minute to hours. (HINT: Reset the lower value when you increment the next one, i.e when you reach 60 seconds minutes goes up and seconds goes to zero.)

Now go back to your main function and add the hours and minutes to the display, see Figure 5.

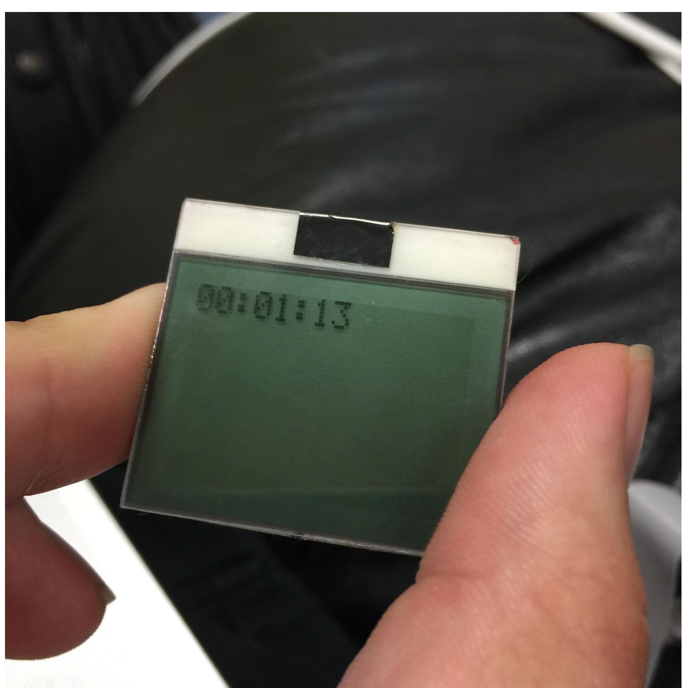


Figure 5: HMS Timer

Assignment 3 Stopwatch

In Assignment 2 you built a run timer that tracks hours minutes and seconds, now you will use that to build a stopwatch. Start by adding a new variable to stm32l1xx_hal.c

```
uint32_t startTick = 0;
```

This variable is used so that you can start the timer at any point, while the function is running. Again you will need a function to get the startTick, call it getStartTick(), but this time you will also need to set the startTick from your main function. The stopwatch can start 1 sec into the application or 12 hours, as such you will need to add the function setStartTick, which should set startTick to the current uwTick.

As you will need to reset seconds, minutes, and hours every time you reset the stopwatch you will also need functions to set each of them (HINT: make one to set each value not just reset it, as this will be useful later). Now adjust the HAL_IncTick so that seconds are calculated using

startTick. (Hint: The difference of uwTick and startTick is the time). Now use the button on the STM32H152 to start and stop the timer, see Lesson 1 Assignment 2 if you have forgotten. (as you only have one button you will have reset the stopwatch on start). I would put a Figure, but it would look identical to Figure 5.

Assignment 4 12 Hour Cycle

In assignment 3 you build a stopwatch that tracks hours minutes and seconds, now you will use that to build a clock. And the AM/PM shift to your stm32l1xx_hal.c, (get, and set functions as well as , and HAL_incTick functions) Now display the time so it starts at midnight (12:00:00AM) and have it keep track of the time from there, see Figure 6.

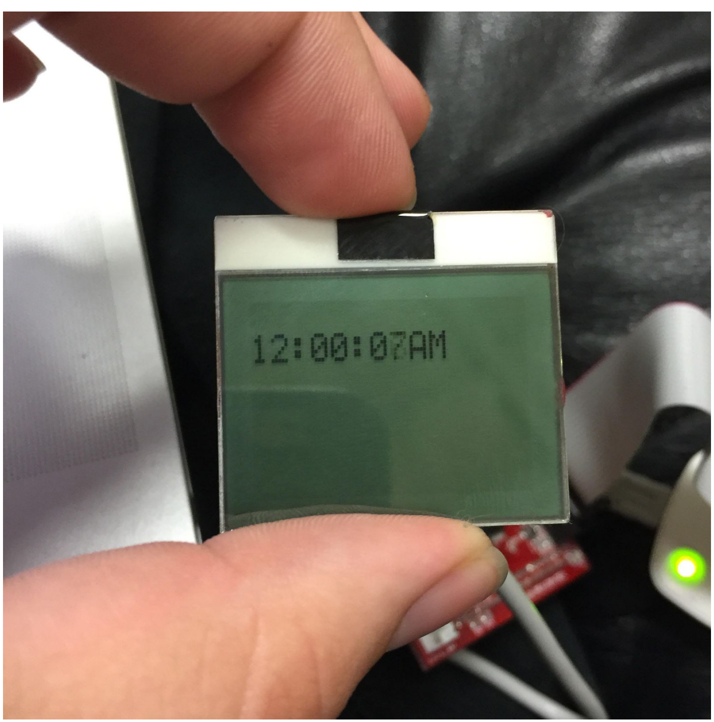


Figure 6: Hour Cycling Clock

Project 2 Clock

You now have a functional time keeper, but unless you start it at midnight it will not display the current time. Remember those setter function you created in assignment 2, now this is where they are useful. You know how to use the keypad from the calculator lesson, use the keypad and the setter function so that you can set the current time, be sure to only allow legitimate times (i.e. 36:92:00PM is not). You do not need to set the seconds have them reset when a new time is entered.

- ❏ Set Hours Minutes and AM/PM
    - ❏ Seconds Reset when the clock is Set
- ❏ Legitimate Times
- ❏ Keeps Time

Bonus Project Multi-Function Clock

-Alarms

-Stopwatch

-Timer

-ect.

Appendix D Morse Code Interpreter Lesson

Lesson 4 Morse Code - Antique Texting

This lesson will teach you how to make a morse code interpreter, through which you will learn how to use interrupts. You will also learn Morse Code, which will help if you're ever stuck on a deserted island.

Need to Know 1 Morse Code

Morse code was invented a way to sent text information over a distance, be it by sound, light, or telegraph. It uses a series of flashes, click, or pulses of varying length to create a message. In Figure 1 you can see the explanation of codes for the alphabet and numbers.

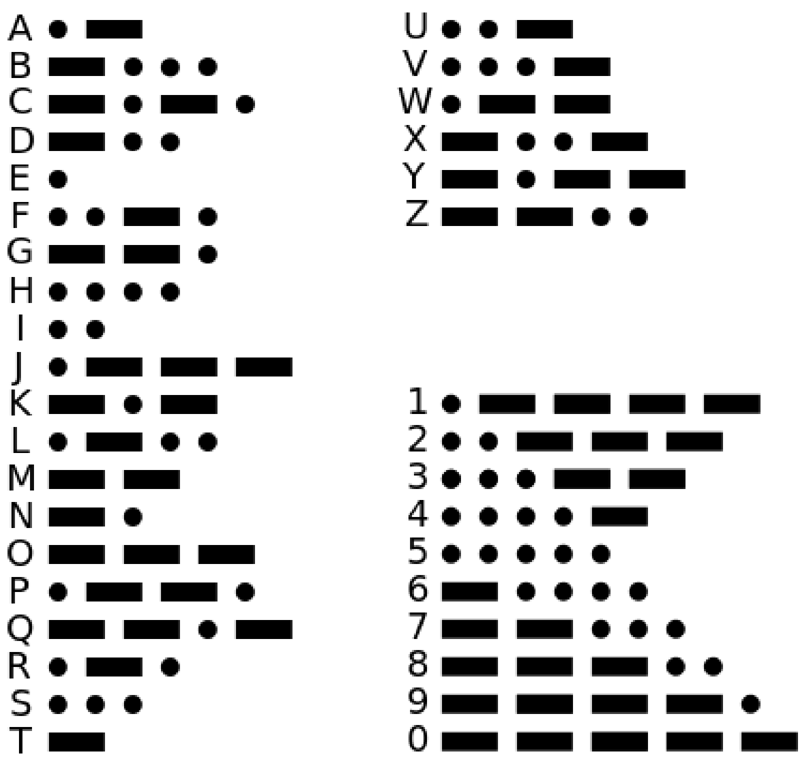


Figure 1: Morse Code Alphanumeric Chart

Need to Know 2 Morse Code Tree

Now the question of how to interpret morse code programmatically, the simple answer is a binary tree. You start at a blank node then make right child a dot and left child a dash, and you get a binary tree.

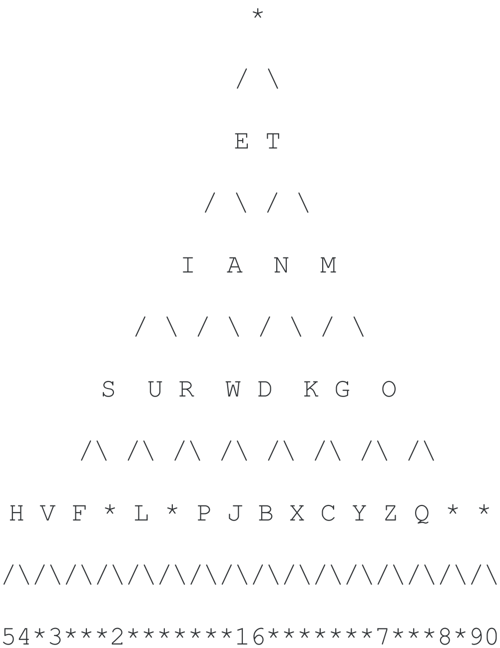


Figure 2: Morse Code Tree

To use this in your code you can cheat and use a string and the principle that given int n where n is a number 1 or greater and string[n] is a character, the right child (dot) can be found at 2n and the left child (dash) can be found at 2n+1. This allows you to use the string:

ETIANMSURWDKGOHVF*L*PJBXCYZQ54*3***2*******16*******7***8*90

To navigate morse code, using n =1 as the first node. (Do not start at n = 0). * indicate an empty node, or a node that contains something not important to your project.

Assignment 1 Keypad Interpreter (Letter)

Now that you have an understanding of how morse code works you can, make an interpreter. Open the example project in the Morse Code folder in STM32H152 Kit. Use the string provided above to make a program that allows you to interpret a code that is entered. Use the keypad to simulate dots, dash, character end. (ex 1 = dot, 2 = dash, etc). I recommend that you display the code one screen as you enter it. It will help remember that the max size for any character is five dashes/dots. When you hit the character end key have the letter appear on the screen. I would also recommend that have a clear function. See Figure 3 for what a finished product should resemble

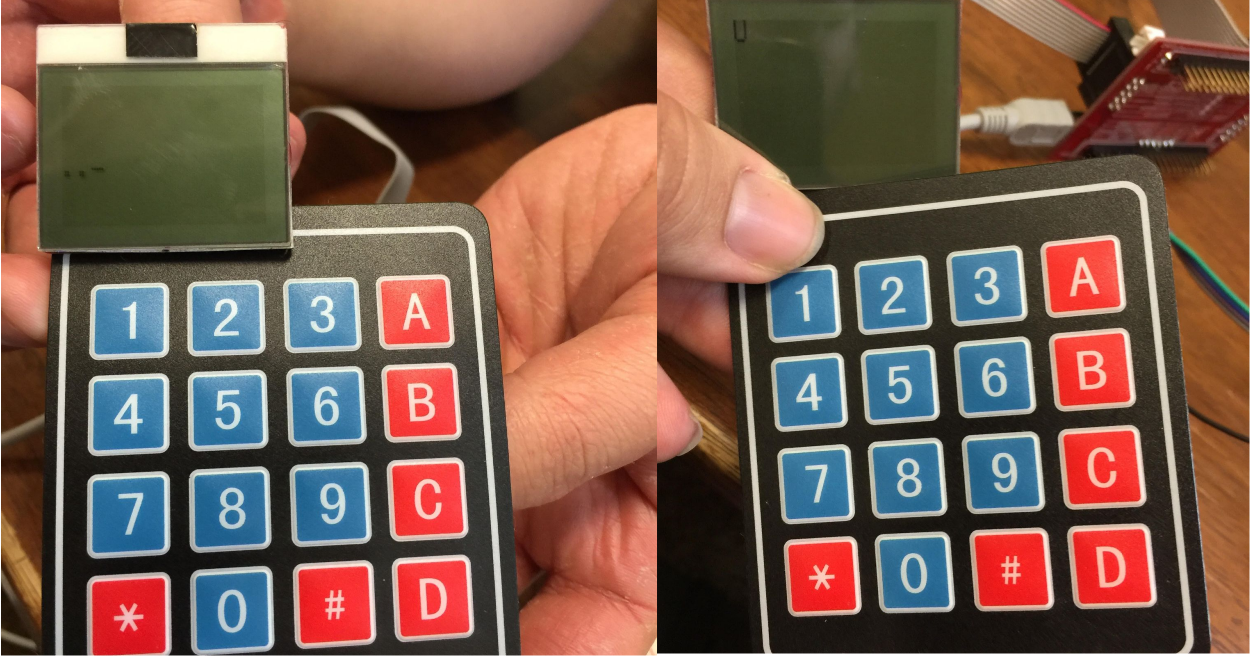


Figure 3: Interpreter

Assignment 2 Keypad Interpreter (Message)

You now have a function interpreter, you just need to add two things and you will be able to enter message, first you must implement a space key, Then you must add a string to hold your message. (Remember to make n+1 where n is your current place in the string '\0' to display it on the LCD correctly). I would for the sake of testing adjust the clear function to clear the code on the first push and the message on the second. Figure 4 contains an image of what a finished interpreter should resemble.

Figure 4: Message

Assignment 3 Button Interrupts

This assignment will teach you about interrupts and how to use them with buttons in the HAL code. Before this you have used the method of polling with your buttons, this is when you at a certain place in the code check to see if the button is pushed. The method you are now

learning is called the interrupt. It name is what it does, an interrupt interjects into code when it is triggered. I will use the morse code interpreter you will be building later as an example, you make the code passively display the current message, and code like in your keypad interpreter. However on the push of a button, the microcontroller pauses the display code and runs the interrupt.

Back in the Lesson 1 there was an assignment that explained how to initialize the button on the Board. The following code (minus the line numbers of course) are how program a button push to trigger an interrupt.   Following the code is a line by line description of what is occuring.

```
1.  __HAL_RCC_GPIOA_CLK_ENABLE();
2.   GPIO_InitTypeDef Buttoninitstruct = {0};
3.    Buttoninitstruct.Pin   = GPIO_PIN_0;
4.    Buttoninitstruct.Pull  = GPIO_NOPULL;
5.    Buttoninitstruct.Speed = GPIO_SPEED_FREQ_HIGH;
6.    Buttoninitstruct.Mode  = GPIO_MODE_IT_RISING;
7.    HAL_GPIO_Init(GPIOA, &Buttoninitstruct);
8.    HAL_NVIC_SetPriority(EXTI0_IRQn, 0x0F, 0);
9.    HAL_NVIC_EnableIRQ(EXTI0_IRQn);
```

1. Enables the clock on the Port A, which is where the Button connect to the microcontroller.
2. Creates a GPIO_InitTypeDef, which is a construct made by the HAL libraries to contain the information needed to initialize pins.
3. Specifies the Pin in Port that the button connects to can be verified in the circuit diagram.
4. Specifies on whether the button requires a pull up/down resistance to function
5. Sspecifies the refresh rate of the pin

6. This line is the most important, the mode up you used in Lesson 1 and the keypad driver both use GPIO_MODE_INPUT, which enable passive listening (it tracks the value and you can access it when needed). GPIO_MODE_IT_RISING which is the mode used in this example means an interrupt will be triggered on the rising edge, meaning when you push the button down. There are two other interrupt modes GPIO_MODE_IT_FALLING (button release) and GPIO_MODE_IT_RISING_FALLING (Push and release both trigger separate interrupts: this one will be important later)
7. The initialization of the pin using the struct
8. Set the priority of the interrupt. priority is the importance of interrupts, interrupts with a higher priority will run before interrupts with a lower
9. This line turns the interrupt on.

Now go open the example workspace in the Button Interrupt folder.

You will see the above code and the initialization of the LED in the main function followed by an empty while (1). At the bottom you will see the function:

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
  BSP_LED_Toggle(LED4);
}
```

This function is the interrupt, it is the code that will run when the button is pushed. In this case it will toggle on of the LEDs on the board. Run it and try. The LED will only toggle on the push of the button.

Now change the mode to GPIO_MODE_IT_FALLING the toggle should occur when you release the button

Now change the mode to GPIO_MODE_IT_RISING_FALLING this should make the LED stay on as long as you hold the button down

Finally, change the mode to GPIO_MODE_INPUT and add the following code to the while(1) loop

```
if (BSP_PB_GetState() == GPIO_PIN_SET){
	BSP_LED_Toggle(LED4);
	HAL_Delay(200);
}
```

Now push the button, it toggles. Now hold the button, your LED should be blinking this is because it keeps polling the button every 0.2 secs.

Assignment 4 Rise/Fall Interrupts

This assignment will have you use rise/fall interrupts to do separate things. Make it so that on push (rise) LED3 toggled and on release (fall) LED4 is toggled (HINT: Assume that the starting state is not pushed and use a flag)

Project 3 Morse Code Interpreter

Now on to the main part of the lesson, a working morse code translater. See Figure 5 for a picture of the finished project.

- ❑ Pick a speed (remember each tick is 1 ms) referred to as beats below

- ❑ Reads in morse code
  - ❑ Dot (hold button down 1 beats)
  - ❑ Dash (hold button down 3 beats)
  - ❑ Code Space (released 1 beats)
  - ❑ Letter Space (released 3 beats)
  - ❑ Word Space (released 7 beats)
- ❑ -Make certain the Letters appear at apprx. 3 beats
- ❑ -Leave room for error ex dot ~0-2.5, dash ~2.5+
- ❑ -Display message and code like in the keypad interpreter
- ❑ -Use Keypad to add a clear function

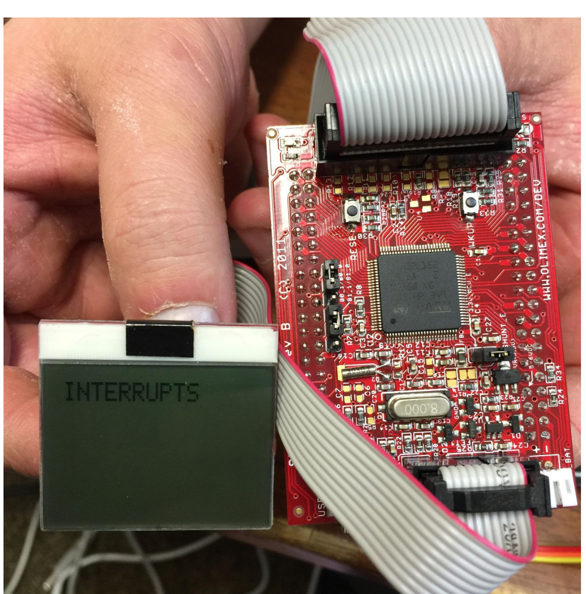


Figure 5:Morse Code Interpreter

Appendix E Maze Game Lesson

Lesson 5 Simply a-MAZE-ing

Figure 0:Maise Maze

In this lesson, you will be learning how to use the drawing functions built into the LCD to design a working maze game. This project will also help reinforce the concept of programming dynamic states within a code. In addition, this will be the first ever project where you will be constructing a game.

Assignment 1: Drawing a line

The most basic shape is a line or line segment. The function "LCDLine" will draw an almost straight line between two points. The two points are set as the parameters for the equation, with the first two parameters being one X-Y coordinate, and the last two parameters being the other X-Y coordinate.

Open up the folder labeled maze game and access the project within. Inside the main function, input the following code:

```
int main(void){
	HAL_Init();
	LCDInit();
```

```
    LCDContrast(0x70);
    LCDLine (1, 1, 83, 47);
    return 0;
}
```

This code should in theory draw a line segment from the upper-left-hand corner of the board to the lower right-hand-corner. Once you have formatted the code, download the program to the board, and debug it.

Despite your best efforts, the board is completely blank. You see, the functions that you will be using to draw different objects do not actually write on the board; at least not by themselves. What happens is slightly more complicated.

How it works: Built into the lcd3310_GPIO is an array that acts as a memory bank, simply called "memory". This array has a size that is equivalent to the total number of pixels on the board (48x84, or 4032). And as you might have guessed, each spot corresponds with a particular pixel. The function "LCDPixelXY" takes a particular pixel at some coordinates, and then initializes the spot in the array that matches that pixel. However, this still doesn't explain how the pixel is actually written on the board. For that, we turn our attention to the function "LCDUpdate". When this function is called, the program takes the entire memory array, and sends all the values contained within the array, one at a time, to the LCD scree, which interprets the value of the array as the pixel point potentially being initialized. At every point that is initialized, the pixel on the screen is set to appear as a black dot. Nothing is 'drawn' onto the board until "LCDUpdate" is called.

Now, try writing the code again, and before the return statement, write:

```
LCDUpdate();
```

You should see the line take shape, See Figure 1.

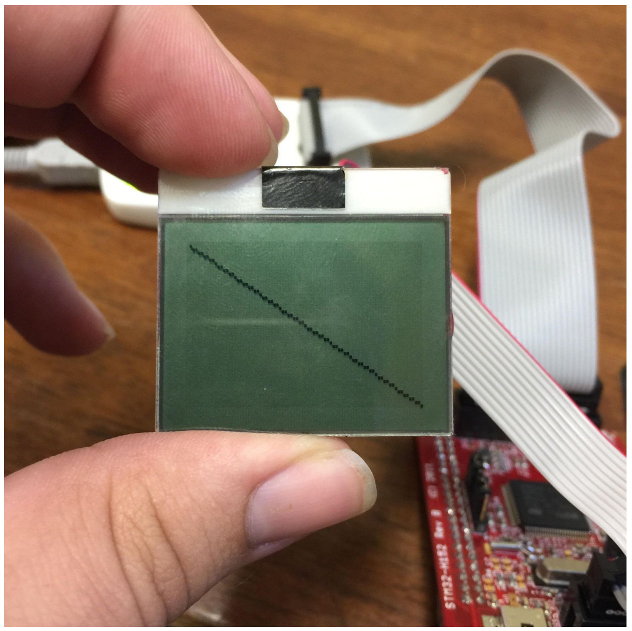


Figure 1: It's a Line!

Assignment 2: A rectangular Box

The function used to draw a rectangular box is "LCDRectangle". This function takes in four unsigned chars, which set the first X and Y coordinates, and the second X and Y coordinates of the box, respectively. The function uses "LCDLine" to create 4 line segments (2 vertical and 2 horizontal), that together form the sides of a rectangular box See Figure 2 for a sample box.

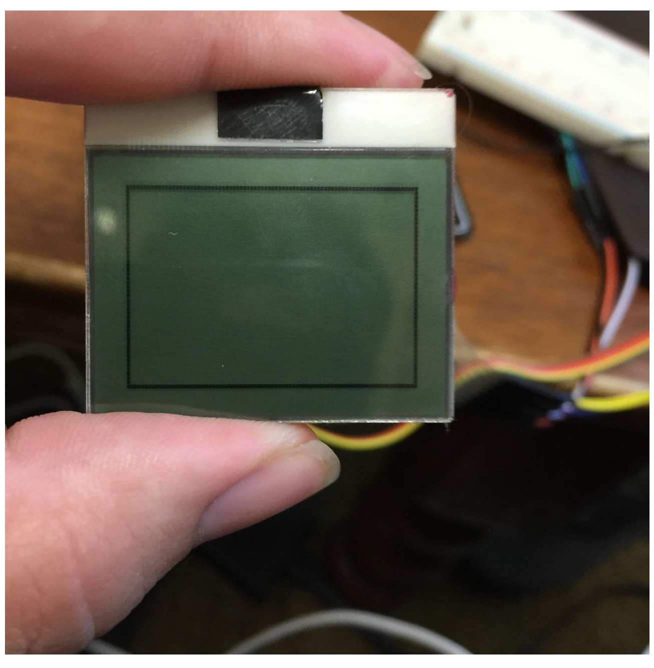


Figure 2: Hopefully no Snakes in this Box

Assignment 3: The Box Moves

Drawing an object is easy enough, but now we need to make our object change position. Let's try moving one pixel down and one pixel right. Simply rewrite the function to draw a rectangle, and add 1 to each of the values. However, if you ran the code now, you'd end up with two rectangles. To simulate movement, you'll need to erase the first rectangle before drawing the second. Thus, it's time to introduce the function "LCDClear". This function will reset all the values in the memory array back to 0, effectively resetting the board. This change will not be reflected on the board until "LCDUpdate" is called. Try moving the box around the screen.
(Note: The movement will work best if you place a delay after each update)

Assignment 4: Circles

Another function used to draw is "LCDCircle". This function takes in 3 parameters: the first two are unsigned chars X and Y that mark the center of the circle, and the third parameter is

the radius of the circle. For now, create a circle with a radius of 2, and set the coordinates to x=5 and y=5., see Figure 3

Figure 3:Tiny Circle

Assignment 5: Freeform movement

At this point, you've been setting the coordinates for the object individually, which can prove to be tedious and is impractical when dealing with more complex movement. For your future assignments, you'll need to develop a system that will keep track of the current location of your object.

Example for Circle: create three variables (for the x-position, y-position, and radius) and set them as unsigned chars of a determinant value. For each movement, the X or Y variable will be changed, and then the circle can be redrawn using the variables.

Assignment 6: Blocked Movement, and States

Because the memory of the LCD screen is an array of a set size, it physically can not draw anything outside of the limits of the board. To avoid such errors, "LCDPixelXY" includes 2 lines of code that check to ensure that the X and Y coordinates being called are not out of the screen's range. These boundaries prevent the code from trying to initialize a point outside the Memory Array, which could cause a fatal error. In a less extreme circumstance, additional boundaries can be set to keep an object from escaping a confined space.

For this next experiment, create an object and set it at coordinates x=10,y=10. Create a function that will constantly increase the x position and update the board to match. Once this is done, your challenge is to create an additional function that will prevent the x-position from increasing beyond 24. (Hint: If you're stuck on how to accomplish this task, we would recommend creating two individual states for the object: One state for when the object is moving, and one state

Project 4 Make a Maze

In this project you will create a navigable maze, and have player solve your maze by moving a representation of themselves through it (using the Keypad as a D-Pad) , see Figure 4 for an example of a simple maze.

- ❏ Create a circular object to represent the player
- ❏ Use buttons/keys to allow player movement
- ❏ Use lines to draw a maze on the LCD, with set boundaries
- ❏ Update the Movement of the marble every time the key is pressed
- ❏ Stop the marble from moving past set boundaries

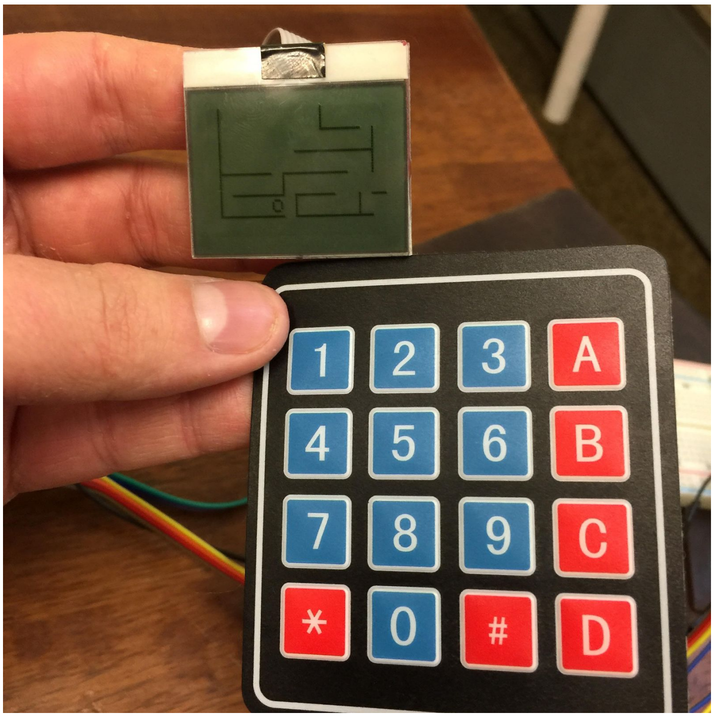


Figure 4: Simple Maze

There are also a selection of bonus challenges that you can use to personalize your application. These are mostly things you can add for fun and they may require more thought and planning to finish.

- ❏ Try displaying the object as something other than a sphere
- ❏ Have the player object face the direction of movement
- ❏ Create an even more complex maze
- ❏ Develop a simplified method for creating lines and boundaries
- ❏ For an extra cool challenge, try adding a game state for collecting a key to open the door to the exit (You'll need to build two separate mazes: one with the blocked exit and key

Appendix F Simon Says Lesson

Lesson 6: Simon Says….Let's Make a Simon Says

This lesson will teach you to write the basic driver for a module, in this case the User I/O module contained in your kit. You will then use your own driver to create a simple game of simon says using the buttons and LEDs on the module.

Assignment 1 LEDs

Open the Simon Says example workspace in the STM32H152 Kit files. This should open you to the file "IO.h". This header file contains the information you need to build the driver for the I/O module as well as containing some helpful Type_Defs and macros that you should use in the construction of the driver. It also contains the prototypes for the functions you will need to create in the driver.

Next you need to open a new blank document, 🗋 . Save it in the folder STM32H152 Kit/Drivers/Modules/Src as "IO.c". Next go to the file explorer on the left side of the IDE, Expand the Drivers, then the Modules, then the Src groups. Right click on the Src group select add file and add I-O.c, see Figure 1.

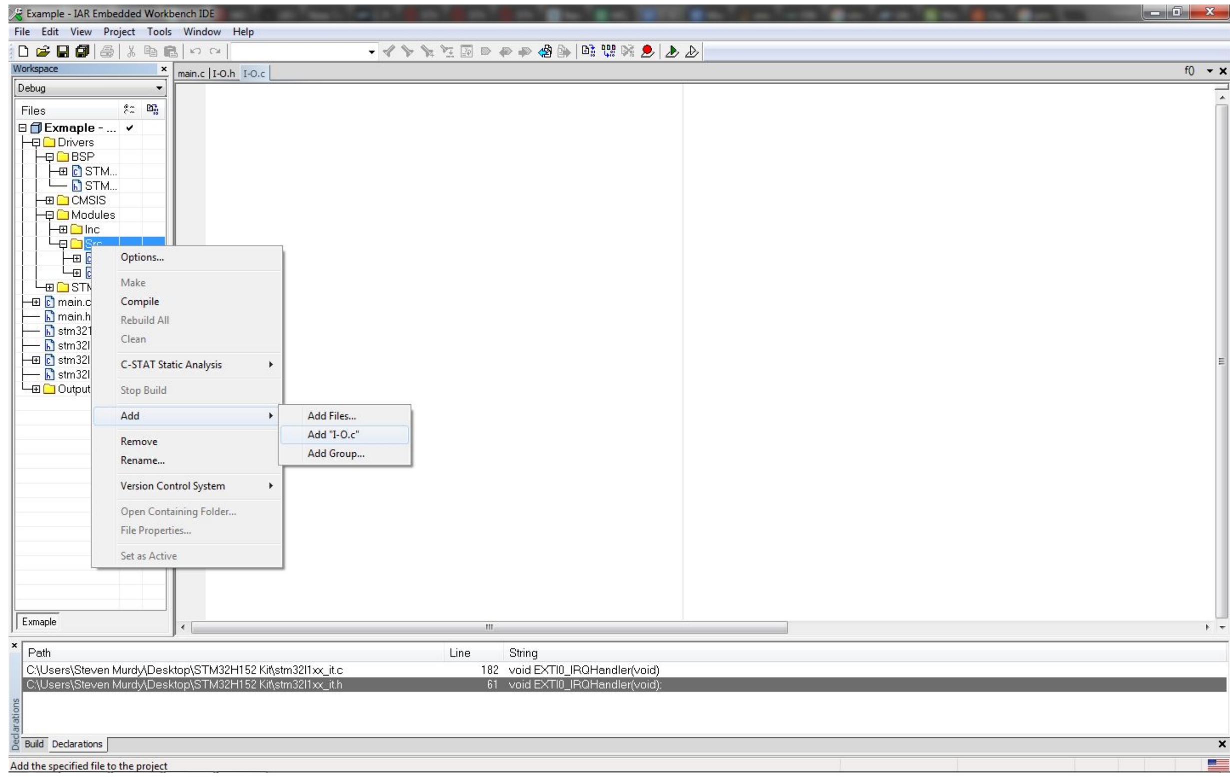


Figure 1:Add your new File to the Workspace

Now add the line:

```
#include "I-O.h"
```

Now I recommend using keypad.c as example of good commenting and how to set up some of the functions. The next step is to use what you know from the previous lessons, to create the functions for LEDs which are prototyped in the header file:

```
void    IO_LED_Init(IOLed_TypeDef Led){
//(NOTE: The LEDs need to be set to open drain, and remember your clocks!!)
//(HINT: Init structs are nice)
}

void    IO_LED_On(IOLed_TypeDef Led){
//(HINT: PIN_SET Off is on and On is Off)
}
```

```
void      IO_LED_Off(IOLed_TypeDef Led){
//(HINT: PIN_SET Off is on and On is Off)
}

void      IO_LED_Toggle(IOLed_TypeDef Led){
//(HINT: Isn't there a HAL for this?)
}
```

Now got to main.c we took care of the HAL_Init, now you need to test your functions, blink some LEDs.

Assignment 2 Buttons

Your next assignment is to create the functions for the buttons, so go back to I-O.c and add the functions:

```
void      IO_PB_Init(IOButton_TypeDef Button, IOButtonMode_TypeDef Mode){
(HINT: There are 4 modes that you will need Input and  Interrupt: Rising, Falling, Rise/Fall.)
}

uint32_t  IO_PB_GetState(IOButton_TypeDef Button){
(HINT: isn't there a HAL function for this)
}
```

Now repeat your examples from assignment 2.1 from the Morse code lesson to test your driver. Set up all four at once one per button that has a corresponding LED.

Project 5 Simon Says

Now use your driver to make a simon says game. Here is a checklist for you

- ❏ Randomly chooses pattern
- ❏ Set Time to press button
- ❏ Win State
- ❏ Lose State
- ❏ Idle State (between games)

(HINT: Flags are nice)

(NOTE: we ran into some problems with IAR's implementation of rand() so here is some code to help)

```
 srand((unsigned) HAL_GetTick());
   HAL_Delay(1);
   if (HAL_GetTick()%2 == 0){
     HAL_Delay(5);
     srand((unsigned) HAL_GetTick());
   }else{
     for(int i = 0; i < rand()%10000; i++){
```

```
    }
     srand((unsigned) HAL_GetTick());
   }
  /*Your Variable here*/ = (rand()/7)*HAL_GetTick() % 4;
```

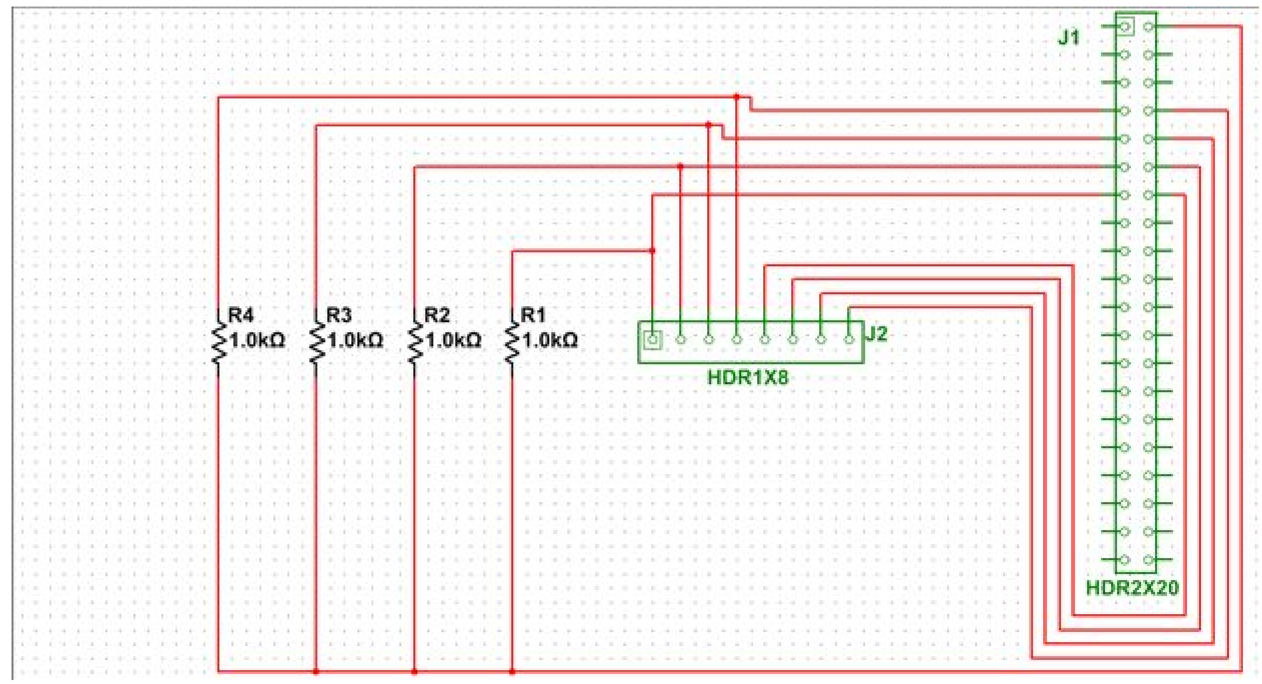
Congratulations, you have reached the end of your kit, but there is much more to learn. We have started you on your path, and hope you continue, we could use the help-

Steve and Alex

P.S. Have some cake

Figure 2: Cake

Appendix G Keypad Module Circuit Diagram



Appendix H User I/O Module Circuit Diagram